Multilevel Parallel Patterns

Rohan Kumar

Supervisor: Dr Christopher Brown

School of Computer Science
University of St Andrews

This dissertation is submitted for the degree of
Bachelors of Science

March 2024

Abstract

Parallel patterns or skeletons are high-level abstractions that simplify programming for parallel hardware systems. While there are several existing parallel pattern libraries that target shared-memory multicore processors such as Intel Thread Building Blocks or distributed-memory architectures utilizing MPI, many high-performance systems involve distributed nodes with multicore processors. This project aims to develop a prototype pattern library that implements basic parallel patterns (such as farm and pipeline) using a multilevel approach to distribute work between the distributed level and the node level. The distributed level aims to maximize parallelism by utilizing a cluster of multicore processors, while the node level exploits parallelism through shared-memory techniques across all cores on each multicore processors in the cluster. The thesis will focus on the design and implementation of the parallel pattern library and evaluate its performance on various parallelizable problems.

Declaration

Rohan Kumar
March 2024

Table of contents

List of figures

List of tables

Chapter 1

Introduction

This project presents the design and implementation of a prototype parallel patterns library using C++. The library aims to facilitate the development of efficient parallel applications by providing a set of commonly used parallel patterns, such as the Farm and Pipeline patterns that can be used both at the Node and Distributed levels.

1.1 Motivation

Over the past few decades, there has been significant growth in the performance of computing systems. This growth has been mainly driven by the continuous increase in transistor densities and clock frequencies, in line with Moore's law [23]. However, the traditional approach of improving performance through higher clock speeds in single-core processors has faced fundamental physical limitations. These limitations include the increased power consumption and heat dissipation associated with higher clock speeds [7], leading to higher costs in cooling these chips and the potential for hardware errors. Additionally, as parts of the chips approach the scale of atoms, they encounter challenges posed by quantum effects [23].

To overcome these limitations and sustain the trajectory of performance improvements, the computing industry has embraced parallel computing architectures, particularly multi-core processors. These processors leverage multiple independent processing cores integrated onto a single chip, enabling parallel execution of tasks and offering enhanced computational capabilities without proportional increases in power consumption.

The transition to multi-core architectures, however, posed significant challenges for software developers. Parallel programming, which is essential for leveraging the full potential of these architectures, is inherently complex, as it introduces issues such as deadlocks, race conditions, and synchronization problems to name a few. The science of parallel programming lagged behind the advancements in parallel hardware [7], necessitating novel approaches to simplify the development of efficient parallel applications.

Parallel pattern libraries emerged as a solution to this challenge. These libraries encapsulate common patterns of parallel computation, abstracting them into reusable components. By providing high-level abstractions and hiding the complexities of parallel programming, these libraries aim to make parallel programming more accessible and productive for developers.

However, considering the existence of established parallel pattern libraries, why embark on building a new one? While many existing libraries implement patterns on either shared memory multi-core architectures or distributed memory architectures utilizing message passing, they often do not leverage both. This project aims to develop a pattern library that operates on both at node and distributed levels, specifically targeted a cluster of distributed multicore processors. this project explores the potential benefits of utilizing nested node-level patterns within distributed-level patterns. The key aim is to assess the advantages of this multi-level approach to utilizing patterns.

1.2 Project Objectives

Below is a list of the original objectives of the project as documented in the Description, Objectives, Ethics & Resources (DOER) document from the first semester.

1.2.1 Primary

1. Design and implement node-level parallelism library using shared-memory threading
    - Utilize a multithreading library such as POSIX threads to implement parallel patterns for fine-grained parallelism across cores within a node.
    - This will include the farm and pipeline pattern.
    - This will involve developing thread-safe data structures and using synchronization primitives to allow concurrent access to data.
2. Design and implement a set of distributed parallel patterns using MPI that enables parallelism across nodes in a cluster
    - Implement the farm and pipeline patterns at the cluster level using MPI communication primitives.
    - This will involve designing load balancing strategies to distribute independent tasks to available nodes.

- Allow the nesting of node-level patterns within the distributed patterns to achieve the core goal of the project which is multi-level parallel patterns.

3. Evaluate the performance of the library
    - This includes testing the library with at least one example problem for each pattern.
    - Compare the performance to other parallel pattern libraries that already exist like Fastflow.

1.2.2 Secondary

1. Add support for at least one other parallel pattern like map at node level.
2. Add support for at least one other parallel pattern like map at cluster level.
3. Evaluate performance of library with two or more example problems for each parallel pattern both at the node and distributed levels.

1.2.3 Tertiary

1. Add library support for at least one other architecture type like a GPU.

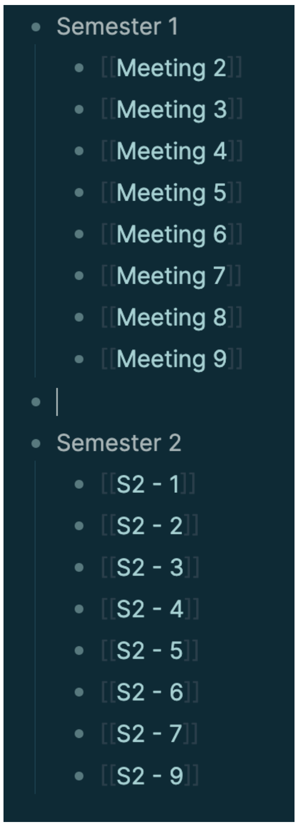
All the primary objectives have been met but the 3rd primary objective is only partially met. There is at least one example problem for each parallel pattern that evaluates the performance of the library but the parallel pattern library developed was not compared to other parallel pattern libraries like FastFlow. The third tertiary object has been met as there are two examples each that demonstrate the performance of the library for farm and pipeline patterns at the node-level.

1.3 Software Engineering Process

Given the nature of the project, which involved extensive research and experimentation with the library being developed, a structured software engineering process was not strictly adhered to. However, the approach adopted drew inspiration from both iterative and waterfall models.

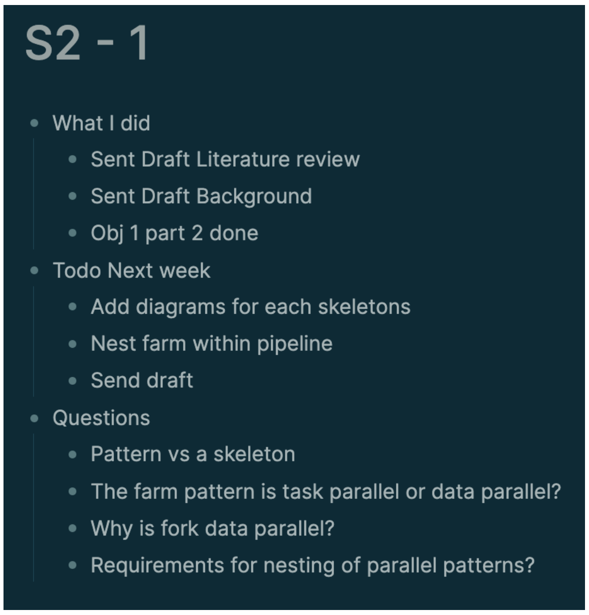
The waterfall model provided a helpful framework for planning the Research, Design, Implementation, and Testing phases of the project, since the requirements set by the project lead were well-defined and stable. Yet, the rigid nature of the waterfall model, which does not allow for client feedback during development, was not entirely suitable for this project. Instead, the iterative model was incorporated to enable ongoing feedback and incremental progress towards the final product.

Ideas from the iterative model is used, which involves multiple iterations of Research, Design, Implementation, and Testing phases. This approach is more open to feedback, allowing the project to evolve gradually towards its final form while producing working code on a weekly basis. By blending these techniques, we create the iterative waterfall model.



- Semester 1
  - [[Meeting 2]]
  - [[Meeting 3]]
  - [[Meeting 4]]
  - [[Meeting 5]]
  - [[Meeting 6]]
  - [[Meeting 7]]
  - [[Meeting 8]]
  - [[Meeting 9]]
-
- Semester 2
  - [[S2 - 1]]
  - [[S2 - 2]]
  - [[S2 - 3]]
  - [[S2 - 4]]
  - [[S2 - 5]]
  - [[S2 - 6]]
  - [[S2 - 7]]
  - [[S2 - 9]]

Fig. 1.1 Meeting notes taken on Logseq every week.



S2 - 1

- What I did
  - Sent Draft Literature review
  - Sent Draft Background
  - Obj 1 part 2 done
- Todo Next week
  - Add diagrams for each skeletons
  - Nest farm within pipeline
  - Send draft
- Questions
  - Pattern vs a skeleton
  - The farm pattern is task parallel or data parallel?
  - Why is fork data parallel?
  - Requirements for nesting of parallel patterns?

Fig. 1.2 Example note taken for Semester 2 Week 1 meeting.

Figures 1.2 and 1.1, outline the notes taken to track progress and improvements in the code each week. These notes include a list of completed tasks, tasks for the upcoming week, and questions for the supervisor. Additionally, a GitHub repository was setup to streamline project code management.

1.4 Ethics

There are no ethical considerations involved in this project, as it does not involve the use of secondary data sets or research with humans and animals.

1.5 Outline

The dissertation begins with a background section in Chapter 2, introducing readers to the concept of parallel patterns, their various types, and the techniques used in their implementation. Chapter 3 offers a comprehensive survey of the context, exploring the landscape and evolution of parallel pattern libraries, as well as examining existing solutions. Chapters 4 and 5 delve into the design and implementation details of the developed library. Chapter 6 provides a critical evaluation of the prototype parallel pattern library, highlighting its strengths and weaknesses. Lastly, Chapter 7 presents conclusions drawn from the research conducted and suggests potential future word that can be done.

Chapter 2

Background

This section aims to provide an introduction to parallel patterns or skeletons (2.1), including an overview of the various types of patterns, common examples for each type of pattern (2.2), and the implementation techniques used to bring them to life (2.3).

2.1 Parallel Patterns

Parallel Patterns, also known as algorithmic skeletons, offer a structured way to represent common forms of parallel computation, communication, and interaction [11, 21]. They are programming constructs that abstract patterns of parallel computation and interaction [22]. These patterns provide programmers with specialized higher-order functions, as proposed by Cole [11], allowing them to craft a tailored solution for a specific parallel problem.

Each pattern can be seen as a template or framework within a parallel programmer's toolbox. By selecting the appropriate template for a given task, the programmer can input the necessary sequential code segments that performs the computation into the template. The end result is a parallel code solution tailored to the specific problem at hand where the coordination part is handled by the library that provides the template. These patterns offer significant value due to their adaptability and versatility.

2.2 Classification of Patterns

Before delving into the different types of patterns, it is important to first define the concept of parallelisable components or tasks. A Parallel Task refers to an independent unit of work that is intended to be executed simultaneously in a parallel computing environment. It operates autonomously, without any dependencies or side effects. Typically, a Parallel Task is encapsulated within a function or a series of statements that can be run concurrently without causing interference. This independence among tasks ensures efficient parallel execution without any contention or synchronization overhead.

Different types of parallel patterns can be categorized based on the functionality they offer. These patterns will be elaborated upon, along with examples of popular skeletons for each type, in the following section.

2.2.1 Data Parallelism

Data parallel patterns are centered around the concept of data parallelism. The fundamental principle is to parallelize a single task by processing independent data elements concurrently. The effectiveness of this approach relies on utilizing stateless process routines, as data elements must be completely independent for efficient parallel processing. In data parallel patterns, the focus is on how tasks are distributed among workers (processors/cores, threads, or machines), as well as aggregating the results of the computations performed by each worker. Some of the well-known patterns in this category include Farm, Map, Scan, and Reduce.

Task Farm Pattern

The Task Farm Pattern involves the parallel execution of multiple worker tasks, where each worker functions as an independent unit of computation. These workers can range from simple tasks to more complex patterns. The Task Farm Pattern does not guarantee the sequential order of execution among the workers. Tasks can be distributed among worker nodes in various ways, and optionally, this pattern can include emitter and collector nodes to distribute tasks to worker nodes and gather results from them, respectively.

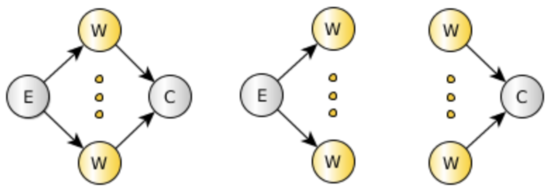


Fig. 2.1 Task Farm Pattern (Source: [1])

In Figure 2.1, three variations of the task farm pattern are illustrated. Each variation includes an Emitter node that assigns tasks to highlighted Worker nodes, which then send results to a Collector node for final compilation. The other variations show scenarios where either the Emitter or the Collector node is absent. The emitter, collector, and worker nodes operate as independent threads, enabling concurrent execution and parallelism.

Map

The Map Pattern entails dividing a task into multiple subparts using a decomposition function. Each subpart is then assigned to a separate worker for parallel execution. Workers independently operate on these smaller units. Upon completion of computations by the workers, a recomposition function combines the results from the subparts to produce the final output. The Map Pattern promotes parallel processing by breaking down a larger task into smaller, independent units that can be concurrently processed. Unlike a farm pattern, in the Map Pattern, the order of results aligns with the order of input problems.

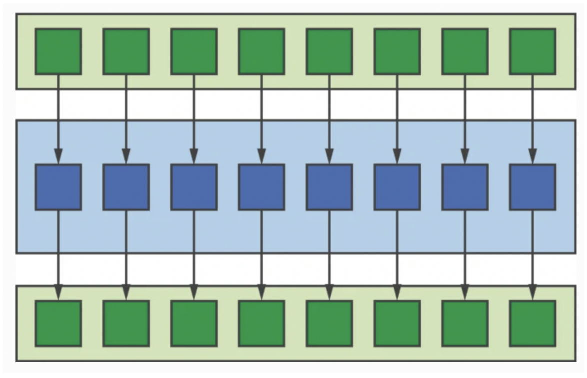


Fig. 2.2 Map pattern (Source: [24])

Figure 2.2 illustrates how the pattern applies the same function (indicated as blue boxes) to map each data elements (represented as the first layer of green boxes say it is labeled by t1, t2, t3 ... tn where n = number of data elements) to its corresponding output elements (represented by the last layer of green boxes t1', t2', t3' ... tn'). Worker nodes are each mapped to different threads to enable concurrent execution, thus introducing parallelism.

Reduce

The Reduce Pattern is commonly employed to efficiently aggregate elements of a sequence and generate a single combined result as output. Reductions typically involve associative actions, such as multiplication, to condense a sequence into a single result [24]. Reduction operations on non-associative functions may yield varying results across different runs. This pattern maintains state to store aggregated partial results and is often used in conjunction with the map pattern for distributed computation, such as calculating the sum of all list elements after doubling each element.

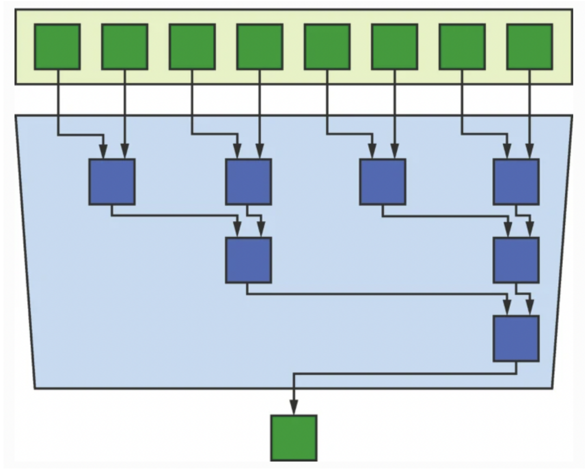


Fig. 2.3 Reduce pattern (Source: [24])

Figure 2.3 showcases a tree reduction pattern [24], demonstrating one specific type of reduction implementation. Reduction operations may combine elements in any preferred order [24]. The figure showcases an array of input elements (in green) being accumulated by multiple accumulators (in blue), with each layer of accumulators processing results from the previous layer. Parallelism is achieved as multiple accumulators are mapped to different threads, enabling concurrent accumulation.

Stencil

The Stencil Pattern is akin to the map pattern but, instead of having access to a single data element, it considers a group of neighboring elements to compute the result [24]. Since the result for each input element can be calculated independently of other elements, data parallelism can be leveraged. This pattern is particularly beneficial for tasks where the transformation or computation of an element depends on the values of its neighboring elements in a grid or array, as seen in image processing operations like Gaussian Blur or Convolution.

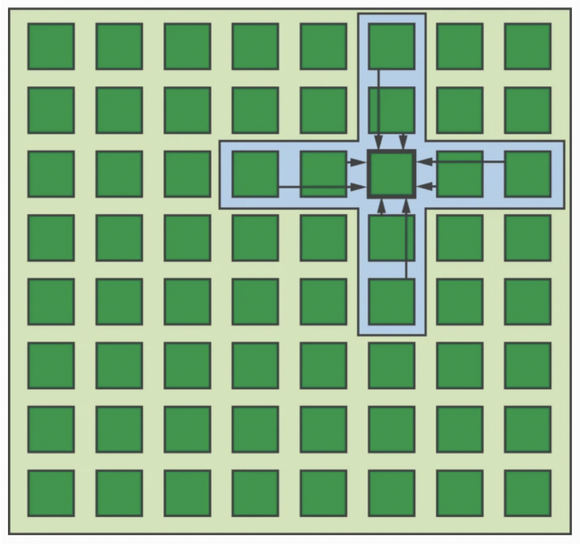


Fig. 2.4 Stencil pattern (Source: [24])

In Figure 2.4, a stencil pattern usage is demonstrated where adjacent elements are utilized to update the data element being processed. The computation for each element typically encompasses a fixed-size neighborhood, as defined by the stencil. In this instance, two elements are considered on the top, left, bottom, and right of the element being processed. The stencil outlines the weights or coefficients used in the computation for each neighboring element. Further details regarding how the computation is mapped to threads will not be discussed, as it is specific to implementation and intricate in nature.

2.2.2 Task Parallel

Task parallelism involves distributing the execution of tasks and coordinating synchronization between them as they run concurrently on different executors (cores or processors). Tasks may have dependencies or a specific order in which they need to be carried out, requiring communication between executors. Different patterns, such as Pipeline and Divide and Conquer, can be utilized based on the level of dependency between tasks.

Pipeline

The Pipeline pattern breaks a task into stages, with each stage performing a specific operation. Tasks move through these stages in a linear manner, with each stage operating independently to allow for higher throughput and parallelism. This pattern is useful for parallelizing tasks like video processing, which involve multiple stages such as decoding, filtering, and encoding.

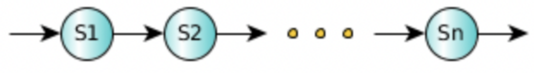


Fig. 2.5 Pipeline Pattern (Source: [1])

Figure 2.5 illustrates a pipeline consisting of n stages. Each stage of the pipeline runs in parallel, typically mapped to different threads that run concurrently.

Fork-Join

The Fork-Join pattern divides a task into subtasks, executes these subtasks independently in parallel, and then combines the results. This pattern is commonly used in recursive algorithms and parallel programming frameworks. It differs from the map pattern in that different functions are applied in parallel to data elements, rather than the same function being applied. An example could be reading from an input stream into a buffer while another thread processes elements in the buffer.

Divide and Conquer

The Divide and Conquer pattern recursively breaks down tasks into subtasks until reaching a trivial task that can be easily solved. To implement this pattern, the user needs to define how to divide a non-trivial task into subtasks, how to solve a trivial task, and how to combine the results of each subtask to obtain the final solution. Sorting a list is a prime example of a problem that can benefit from this pattern.

2.3 Parallel Pattern Libraries/Frameworks

Parallel pattern frameworks furnish a set of parallel patterns or algorithmic skeletons with generic parallel functionality, that are parameterised by the programmer to generate a specific parallel program [21]. Analogous to common software libraries, skeletons are typically accessible through language syntactic extensions or well-defined application programming interfaces [21].

2.4 Parallel Pattern Library Implementation Techniques

In order to implement parallel patterns, it is essential to allocate the components of a parallel pattern (for example the collector, emitter, and worker nodes of a farm) onto distinct computing units (threads or processes) that can operate independently. Effective communication mechanisms must also be established to facilitate interaction among these threads or processes. Parallel Pattern Frameworks rely on two primary programming models to enable this communication: shared-memory and distributed-memory programming.

2.4.1 Shared Memory Programming Model

Shared memory programming enables multiple threads or processes to access a common address space, allowing for parallelized computations. Thread synchronization mechanisms such as mutexes, semaphores, or atomic operations are essential to ensure orderly execution and prevent data races in this concurrent environment. Specialized data structures further enhance the security of shared data handling between threads and processes. Popular shared memory programming models for effective implementation include Intel Threading Building Blocks (TBB) [25] and OpenMP [14]. Additionally, threading libraries in various programming languages can be leveraged to implement shared memory programming techniques efficiently like pthreads of POSIX threads [3].

In C, shared memory parallelism is commonly achieved through POSIX threads [3] and inter-process shared memory communication mechanisms like POSIX shared memory (shm) on Unix-based systems, as well as libraries such as OpenMP [14]. Examples of utilizing these shared memory programming techniques will be explored in the following subsections.

POSIX Threads

Shared memory parallelism can be achieved in C using various methods, one of which is using the POSIX threads (Pthreads) library [3]. Pthreads allows the creation of user-level threads within C programs and provides developers with precise control over thread creation, synchronization, and communication [8].

The pthread library provides mechanisms for thread synchronization and coordination. This includes mutexes for locking access to critical sections of code or data, semaphores for managing access to limited shared resources, barriers for synchronizing the arrival of threads at a specific point, and condition variables for signaling other threads to coordinate on state changes [3].

```c
#include <pthread.h>
```

```
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h> // For sleep function

#define NUM_THREADS 4

// shared resource
int resource = 4;

// mutex protecting resource
pthread_mutex_t mutex;

// limit access to resource
// via this semaphore
sem_t resource_sem;

// barrier for synchronization
pthread_barrier_t barrier;

// producer signals resource available
// on this condition variable
pthread_cond_t resource_cv;

void* thread_routine(void* thread_id) {

    // synchronize on barrier
    pthread_barrier_wait(&barrier);

    // access resource
    pthread_mutex_lock(&mutex);
    while(resource > 0) {
        resource = resource - 1;
        printf("Thread %d took resource\n", *(int*)thread_id);
        sleep(1); // use resource
    }
    pthread_mutex_unlock(&mutex);

    // signal resource freed
    pthread_cond_signal(&resource_cv);

    // synchronize on barrier again
    pthread_barrier_wait(&barrier);

    pthread_exit(NULL);
```

```c
}

int main() {

    // mutex and barrier initialization...

    // initialize binary semaphore
    sem_init(&resource_sem, 0, 1);

    // condition variable init
    pthread_cond_init(&resource_cv, NULL);

    pthread_t threads[NUM_THREADS];
    int ids[NUM_THREADS];

    for(int i = 0; i < NUM_THREADS; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, thread_routine, &ids[i])
            ;
    }

    // join all threads
    for(int i = 0; i < NUM_THREADS; i++){
        pthread_join(threads[i], NULL);
    }

    // destroy synchronization objects
    sem_destroy(&resource_sem);
    pthread_cond_destroy(&resource_cv);
    pthread_barrier_destroy(&barrier);
    pthread_mutex_destroy(&mutex);
}
```

Listing 2.1 C Pthreads Example

Code example 2.1 demonstrates thread synchronization in C using pthreads, semaphores, condition variables, and barriers. A global variable named **resource** is utilized to represent a shared resource among threads. The **mutex** is employed to facilitate exclusive access to this shared resource, while a binary semaphore named **resource_sem** may be utilized to restrict access to the resource, although it is not explicitly utilized in this particular example. The **thread_routine** function shows a common pattern for accessing the shared resource: threads are stopped at the barrier to initiate concurrent execution, acquire the **mutex** to safely decrement the resource, notify other threads once the resource has been utilized, and then synchronize again at a barrier before termination. In the **main()** function, synchronization primitives are initialized, threads are created to execute the **thread_routine**, and the program waits for all threads to finish execution. Finally, all synchronization objects are destroyed to clean up the resources. This example illustrates the management of shared resources and the synchronization of thread execution to prevent race conditions and ensure orderly access to shared resources.

While Pthreads allow multi-threading which is one way of doing shared memory parallel programming where each thread shares the same address space and resources like file pointers, using multiple processes is another approach. Processes have separate private address spaces by default. C provides inter-process communication (IPC) mechanisms like shared memory [4] and pipes [2] to coordinate processes. Shared memory allows direct access to common physical memory pages mapped into the virtual address spaces of multiple processes [26]. Shared memory facilitates tightly-coupled parallelism between processes.

OpenMP

OpenMP [14] (Open-Multi-Processing) is a portable API for writing shared memory parallel programs in C, C++, and Fortran. It uses a directive-based approach to parallelism, with compiler directives and runtime library routines to specify parallel regions, work sharing, synchronization, and data environment. The basic unit of parallelism in OpenMP is a parallel region defined by the #pragma omp parallel directive, which is executed by a team of threads managed by the OpenMP runtime. Within parallel regions, work can be divided among threads through constructs like **#pragma omp** for to parallelize loops, with automatic work scheduling. Synchronization is handled through directives like **#pragma omp barrier** for implicit synchronization and **#pragma omp critical** for mutual exclusion. Data scoping within parallel regions is managed using clauses like private, shared, and reduction.

The creators of OpenMP highlight two main advantages of the library compared to other programming models such as pthreads and MPI. One key strength is its support for **incremental parallelization** [14], allowing programmers to gradually transition parts of their code to parallel execution. This feature is particularly beneficial for parallelizing large legacy scientific research codebases. Another core strength is OpenMP's support for **data parallelism** [14], which is lacking in pthreads. OpenMP provides directives for parallel regions, work sharing, synchronization, and more, simplifying the process of writing data-parallel code. Unlike pthreads, which requires explicit threading and synchronization calls, OpenMP offers higher-level language constructs that make parallel programming less challenging for developers.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define TOTAL_POINTS 1000000

int main() {
    int num_points_inside_circle = 0;
    double x, y;

    // Set the random seed
    srand(omp_get_wtime());

    #pragma omp parallel for private(x, y) reduction(+:
       num_points_inside_circle)
    for (int i = 0; i < TOTAL_POINTS; i++) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;

        // Check if the point (x, y) is inside the circle
        if (x * x + y * y <= 1)
            num_points_inside_circle++;
    }

    double pi_estimate = 4.0 * num_points_inside_circle /
       TOTAL_POINTS;

    printf("Estimated value of PI: %f\n", pi_estimate);

    return 0;
}
```

Listing 2.2 Computing PI through MonteCarlo simulations with OpenMP Reduce

In Example 2.2, OpenMP is utilized to parallelize a program aimed at calculating the value of π using Monte Carlo simulations. This implementation simplifies the need for complex control or synchronization mechanisms. The sequential loop responsible for computing π is parallelized using OpenMP pragmas. The directive `#pragma omp parallel` establishes a parallel region designed to be executed by multiple threads. Within this region, the work-sharing construct `#pragma omp for` distributes the loop iterations among threads. By default, the loop index variable `i` is kept private to each thread. The clause `reduction(+:num_points_inside_circle)` specifies that the partial sums calculated by each thread should be combined into the global variable `num_points_inside_circle` through summation. This eliminates the need for manual synchronization to merge partial results. The OpenMP runtime manages the creation of thread teams, iteration scheduling, and reduction operations seamlessly. With just two pragmas, the computationally intensive loop is parallelized across cores, resulting in improved program performance.

C++11 Threads

The C++11 standard brought significant enhancements to support shared memory parallel programming through its native threads library [13]. This library eliminates the need for external threading libraries like pthreads, providing a standardized, platform-independent approach to multi-threading. Below are some key components of the threads API:

- **std::thread**: Represents a thread of execution created by passing a callable object (function, functor, lambda). It enables portable thread management without external dependencies, facilitating concurrent task execution.

- **std::mutex**: Provides mutual exclusion locking to protect shared data and prevent race conditions. It ensures that only one thread can access critical code sections at a time, promoting safe updates to shared resources.

- **std::condition_variable**: Allows threads to wait for notifications from other threads before proceeding. It enhances synchronization by efficiently handling state changes or resource availability signals.

- **std::async**: Executes a callable asynchronously and returns a std::future for result retrieval. It simplifies asynchronous programming, enabling parallel task execution and easier management of asynchronous operations.

- **std::atomic<T>**: Supports thread-safe operations on an atomic data type T, eliminating the need for locks in certain scenarios. This improves performance and ensures atomicity of shared variable operations without explicit locking.

Apart from these components, C++11 threads also offer utilities such as thread local storage, locks, barriers, and more. The library presents a comprehensive and standardized framework for shared memory parallel programming in C++ [13].

```
#include <iostream>
#include <thread>
#include <mutex>
#include <atomic>
#include <condition_variable>
#include <future>

std::mutex mutex;
```

```cpp
std::condition_variable cv;
std::atomic<int> counter{0};

void incrementWithMutex() {
    std::lock_guard<std::mutex> lock(mutex);
    ++counter;
}

void incrementWithAtomic() {
    counter.fetch_add(1, std::memory_order_relaxed);
}

void incrementWithConditionVariable() {
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock, [] { return counter % 2 == 0; }); // Wait for
        even counter value
    ++counter;
    cv.notify_one(); // Notify waiting threads
}

int main() {
    // Using std::thread
    std::thread t1(incrementWithMutex);
    std::thread t2(incrementWithMutex);
    t1.join();
    t2.join();

    // Using std::atomic
    std::thread t3(incrementWithAtomic);
    std::thread t4(incrementWithAtomic);
    t3.join();
    t4.join();

    // Using std::condition_variable
    std::thread t5(incrementWithConditionVariable);
    std::thread t6(incrementWithConditionVariable);

    // Allow the first thread to increment before starting the
        second thread
    std::this_thread::sleep_for(std::chrono::milliseconds(10));

    cv.notify_one(); // Start the second thread
    t5.join();
    t6.join();

    std::cout << "Final Counter Value: " << counter << std::endl;

    return 0;
}
```

Listing 2.3 C++11 Threading Example

Example 2.3 demonstrates a shared counter protected by different synchronization mechanisms. Three functions (`incrementWithMutex`, `incrementWithAtomic`, and `incrementWithConditionVariable`) increment the counter using `<mutex>`, `<atomic>`, and `<condition_variable>`, respectively. Threads are spawned using `std::thread` to concurrently increment the counter. `incrementWithMutex` uses a `std::mutex` to ensure mutually exclusive access to the counter. `incrementWithAtomic` utilizes `std::atomic` to perform atomic increments without explicit locking. `incrementWithConditionVariable` employs a `<condition_variable>` to synchronize thread execution based on the counter's value.

2.4.2 Message Passing Programming Model

Message passing programming involves parallelising computations by allowing many different processes communicate via immutable-messages that are transmitted over a communication channel or the computer network. This is a common way to program distributed memory systems where processes do not share common memory that can be used for communication. This allows machines with heterogeneous architectures to function as a single machine with a distributed processor. Popular message passing libraries or standards include Message Passing Interface (MPI) [10], and the Parallel Virtual Machine [27]. Here's the updated version with the removed before [**?**]

MPI

Traditionally, lots of multi-processor systems did not support hardware cache coherence[14], which is the problem where multiple caches corresponding to different cores store copies of the same data, and modifying one does not change the other caches. This led to the development of the message passing model of computation, where multiple processors, possibly with heterogeneous architectures, operate on memory distributed across different computing units.

The Message Passing Interface (MPI) has become the dominant model for distributed memory parallel programming in high-performance computing over the last three decades[10].

In the early 1990s, the MPI Forum, a collaborative effort led by Oak Ridge National Lab and Rice University, brought together parallel computing vendors, researchers, and users[29]. Their goal was to define a common, portable standard for message passing, replacing proprietary parallel programming libraries. After extensive discussions, the MPI Forum published the first draft of the MPI standard in 1993[10], which was refined based on public feedback, ultimately leading to the release of the official MPI 1.0 specification paper in 1994[28]. This paper defined the syntax and semantics of MPI communication primitives like point-to-point send/receive operations and collective communication routines like broadcast, gather, scatter, and reduce. The standard was specified in a platform-independent manner, enabling efficient implementations across various parallel architectures. Open-source MPI implementations like MPICH played a crucial role in promoting the standard's adoption by making it freely available on different platforms[10].

The MPI collectives API is the most important part of MPI, facilitating data flow and coordination between processes in a parallel application. It abstracts away low-level details of message passing, enabling developers to focus on the computational aspects of their parallel programs. MPI collectives refer to a set of operations that involve all processes in a communicator, requiring the participation of all processes within a specified group (communicator) to complete. MPI collectives are used to perform common tasks in parallel programming, such as data distribution, data collection, synchronization, and computational operations across processes. Some of the collectives are listed as follows:

1. **MPI_Bcast**: Broadcasts a message from one process to all other processes within a communicator.

2. **MPI_Scatter**: Distributes distinct pieces of data from one process to all processes in a communicator.

3. **MPI_Gather**: Gathers data from all processes to a single process.

4. **MPI_Allgather**: Gathers data from all processes and distributes it to all processes.

5. **MPI_Reduce**: Applies a reduction operation (such as sum, max, min) on data from all processes and collects the result to a single process.

6. **MPI_Allreduce**: Similar to MPI_Reduce, but the result is distributed back to all processes.

7. **MPI_Barrier**: Synchronizes all processes in the communicator, not allowing any process to proceed until all have reached the barrier.

8. **MPI_Scan**: Performs a prefix reduction on data distributed across processes.
9. **MPI_Alltoall**: Sends data from all to all processes, with each process sending a distinct piece of data to every other process.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define TOTAL_POINTS 1000000

int main(int argc, char **argv) {
    int rank, size;
    int num_points_inside_circle = 0;
    double x, y;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Seed for random numbers
    srand(rank);

    // Compute points inside the circle
    for (int i = 0; i < TOTAL_POINTS / size; i++) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;

        if (x * x + y * y <= 1)
            num_points_inside_circle++;
    }

    // Reduce the total count of points inside the circle
    int total_inside_circle;
    MPI_Reduce(&num_points_inside_circle, &total_inside_circle, 1,
         MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double pi_estimate = 4.0 * total_inside_circle /
           TOTAL_POINTS;
        printf("Estimated value of PI: %f\n", pi_estimate);
    }

    MPI_Finalize();

    return 0;
}
```

Listing 2.4 Listing

In the provided MPI C code example 2.4, a parallel computation of π using the Monte Carlo method demonstrates the distribution of tasks across multiple processes with MPI. Each process initializes its random seed using its rank and calculates a local estimate of π based on the Monte Carlo method. This method involves generating random points within a square and determining whether they fall within a quarter circle, approximating the value of π. The number of points generated is divided evenly among the processes. After each process computes its local estimate, MPI's `Reduce` function aggregates these estimates using `MPI_SUM` to compute a global estimate of π. Finally, the master process (rank 0) prints the global estimate of π to the console. This approach enables parallelization of the Monte Carlo simulation, allowing for efficient computation of π across multiple processors.

2.5 Summary

This chapter provides an overview of parallel patterns, which are programming constructs that abstract common patterns of parallel computation, communication, and interaction. It discusses the different types of parallel patterns, including data parallel patterns like Farm, Map, Reduce, and Stencil, as well as task parallel patterns like Pipeline, Fork-Join, and Divide and Conquer. The chapter also explores the implementation techniques used to realize these patterns, such as shared memory programming models like POSIX Threads, OpenMP, and C++11 Threads, as well as the message passing programming model through use of the Message Passing Interface (MPI). These patterns and implementation techniques facilitate the development of efficient parallel applications by providing high-level abstractions and encapsulating the complexities of parallel programming, enabling developers to leverage the full potential of modern multi-core and distributed computing architectures.

Chapter 3

Related Work

Parallel patterns, also known as algorithmic skeletons, have become widely recognized as an effective approach for developing parallel programs [15]. These patterns abstract common forms of parallel computation, communication, and interaction into reusable components that simplify parallel programming. Since the pioneering work on skeleton-based parallel programming in the late 1980s and early 1990s [11], there has been extensive research into developing parallel pattern libraries and frameworks for various contexts.

This literature review will provide an overview of the landscape of research related to parallel patterns over the past few decades. First, it will go through the idea behind structured parallelism with the introduction of parallel pattern libraries. Then we see a brief history and evolution of the development of these parallel pattern libraries, and the benefits and drawbacks of using parallel pattern libraries for writing parallel programs. Finally, this analysis will survey the existing parallel pattern libraries out there by looking at a few programming languages that include C and C++.

By synthesizing key developments in these areas, this review aims to assess the state of parallel patterns research, discuss interesting design and implementation decisions of various parallel pattern libraries, and hopefully draw insights from this to motivate the development of the parallel pattern library being developed as a part of this dissertation.

3.1 Structured Parallelism and the Evolution of Parallel Patterns

Structured parallel programming refers to models where concurrency is expressed through the structured composition of parallel components that represent common parallel computation patterns [15]. This concept emerged in the 1990s with the introduction of algorithmic skeletons, initially proposed in Cole's PhD thesis from the late 1980s [11, 15]. The goal was to streamline complex parallel programming by separating computational concerns from coordination [15]:

- **Computation**: Application programmers choose and combine appropriate skeleton building blocks to represent the parallel behavior of their programs. - **Coordination**: System programmers develop efficient, reusable implementations of these composable skeletons.

Since its inception, various parallel programming frameworks have been developed and put into practice to support this concept [21, 15]. These frameworks offer sets of algorithmic skeletons with generic parallel functionality, customized by programmers to create specific parallel programs [21]. Some frameworks even allow nesting patterns to enhance the parallelism of individual patterns. By using a framework for parallel coding, programmers can provide abstract descriptions and essentially program in a high-level language, with the framework converting these descriptions into operational parallel code. This approach effectively separates the computational aspect of the program (handled by the programmer) from the required coordination between different parts of the parallel program (managed by the framework) [21]. Embracing this structured approach to parallel programming has advantages and limitations that need to be considered.

3.2 Advantages and Disadvantages of Utilizing Parallel Patterns

When considering the benefits of using parallel patterns, it is important to highlight the advantages they offer in terms of providing high-level abstractions. These abstractions empower programmers to articulate intricate parallel computations and workflows in a concise and comprehensible manner, eliminating the need to delve into the complexities of low-level parallelism. This improved clarity expedites the development of parallel programs and streamlines the restructuring of parallel behavior. Furthermore, by encapsulating implementation details within the abstraction, parallel correctness is ensured "by construction," allowing programmers to concentrate solely on the functional logic [15]. The visibility of the parallel structure also facilitates performance portability across various architectures and supports a range of refactoring optimizations [15].

On the other hand, Danelutto et al. [15] identifies some notable drawbacks of skeleton-based programming. The predefined set of skeletons available in frameworks can limit flexibility, as programmers are unable to create new skeletons tailored to their specific requirements. The lack of customizability also restricts the ability to adjust skeleton behavior for individual applications. Additionally, utilizing entirely new languages for parallel programming, rather than integrating libraries into existing languages, can result in a steep learning curve and hinder widespread adoption.

3.3 Historical Development of Parallel Pattern Frameworks

In a study conducted by Danelutto et al (2021) [15], the progression of various parallel pattern frameworks over time and their evolution are outlined. The study identifies three significant phases in the advancement of structured parallel programming models:

1. The "**Pioneering Phase**" (late 1990s): Initial works focused on skeleton algorithm concepts and prototypes to exhibit the feasibility of structured parallel programming. These early implementations were limited in scope, often serving as proofs-of-concept or functional programming prototypes aimed at specific architectures. They had minimal skeleton sets and lacked performance optimizations. Nonetheless, these pioneering works generated interest within the research community.

2. The "**Colonization Phase**" (late 1990s onward): As concepts advanced, comprehensive frameworks emerged capable of providing good programmability, performance, and support for various architectures such as clusters and GPUs. Despite this progress, adoption remained primarily within specialized research groups and communities already engaged in structured parallel programming.

3. The "**Integration Phase**" (late 2000s onward): Several developments facilitated broader integration and adoption of structured parallel programming. Frameworks embraced modern host languages like C++11, expanded to diverse accelerators like FPGAs, and showcased performance improvements on standard benchmarks that garnered attention. During this phase, the adoption of parallel patterns concepts became more widespread, leading many companies in the technology industry to develop their implementations of these libraries.

3.4 Examination of Case Studies

In this section, we will explore examples of parallel pattern libraries in C, C++, and Haskell. Each library discussed will be briefly assessed for its key features and the design choices made by its creators.

3.4.1 C Programming Language

Within this section, we will delve into two prominent parallel pattern libraries compatible with the C programming language: eSkel and SKElib.

eSkel

eSkel [12] is a collection of C functions and type definitions which supplement standard C bindings to MPI with skeletal operations. Rooted in the SPMD (Single Program Multiple Data) distributed memory parallelism model inherited from MPI, eSkel operations are required within a program that has initialized an MPI environment [12]. Developed at the University of Edinburgh, eSkel aims to make parallel patterns more accessible to mainstream application programmers. The key design principles of eSkel, as outlined in the introductory paper by Coles [12], include:

- Minimizing disruption to existing C and MPI programming models
- Enabling integration of ad-hoc and skeletal parallelism by expanding MPI functionalities
- Providing flexibility in pattern semantics to cater to diverse use cases
- Demonstrating advantages compared to traditional parallel programming to encourage widespread adoption of parallel patterns

The *eSkel* library enhances MPI programming by incorporating advanced collective operations, addressing the need for more sophisticated parallel patterns beyond the basic collective functions of MPI like `MPI_Broadcast` and `MPI_Reduce`. It primarily focuses on facilitating the implementation of parallel patterns such as pipelines and task farms crucial in various parallel applications.

Central to *eSkel* functionality are its collective skeleton operations. Each skeleton denotes a collective operation that all processes in an MPI communicator must trigger. This setup allows dynamic grouping and realignment of processes to match the chosen skeleton's semantics, organizing them into distinct activities. These activities could represent stages in a pipeline or workers in a farm. This strategy significantly eases task distribution and collection complexity, as well as management of interactions between different parallel application components. *eSkel* introduces a different communication approach within activities by establishing specialized communicators for safe, context-specific communication support, allowing ad-hoc parallelism within an activity by nesting skeleton calls. Each nested call generates a fresh activity context, enabling processes to reference their current activity's communicator.

The *eSkel* Data Model (eDM) plays a crucial role by bridging the gap between MPI's data handling mechanisms and distributed data's logical view. The model introduces eDM atoms and collections where atoms are MPI triples augmented with a spread tag indicating independent or cohesive data units from processes. Collections represent sequences of these atoms, facilitating specifying input and output data for skeletons. This approach streamlines distributed data management across various parallel applications.

SKElib

SKElib [16] is a C library offering skeletal parallel programming constructs to aid the development of parallel applications on Unix systems. Originating from the University of Pisa, the library's objective is to provide C programmers with accessible parallel patterns utilizing standard Unix mechanisms in a lightweight fashion. SKElib presents a compact set of common parallel skeletons like farm, pipeline, map, and while loop as C functions that can be combined and nested. Skeletons are defined by enveloping sequential C functions with SKElib skeleton functions such as SKE_FARM and SKE_PIPE.

Parallel execution in SKElib follows an SPMD model, with the library managing processes created on cluster nodes and their communication. When a skeleton call is initiated, the library employs **rsh** to launch processes implementing the skeleton templates on designated nodes. These processes communicate via **TCP** sockets managed by the library. This SPMD approach allows reusing the same program code across nodes, with node behavior determined by command line parameters.

A notable advantage of this model is the seamless integration potential of skeleton parallel constructs and ad-hoc Unix parallelism within the same program. Programmers can leverage skeletons for common patterns and resort to standard Unix mechanisms like threads and pipes for personalized parallelization requirements. SKElib skeletons exchange inputs and outputs through standard Unix files. Programmers designate file names when calling a skeleton, enabling them to utilize Unix I/O tools like pipes and redirection for incorporating skeletons into broader

3.4.2 C++ Parallel Pattern Libraries

Within this section, we will explore two prominent parallel pattern libraries in C++: Fastflow, SkePU, and SKElib.

Fastflow

Fastflow [6], created in 2011 by Aldinucci et al., is a programming framework designed to optimize the utilization of cache-coherent shared-memory multicore systems [6]. The primary focus of Fastflow is on stream parallelism, a programming paradigm that enables the parallel execution of a stream of tasks across sequential or parallel stages. A stream program is represented as a graph of independent stages (kernels or filters) that communicate through data channels. The concept of streaming computation involves applying a sequence of transformations to data streams in the program. Each stage reads tasks from the input stream, processes them, and outputs tasks to the output stream. Parallelism is achieved by executing each stage concurrently on subsequent or independent data elements. Stages may contain local state individually or distribute it across streams. Initially designed for shared-memory multicore systems, Fastflow was later expanded in 2012 to support distributed systems, allowing parallel patterns to operate across clusters of multicore workstations [5].

Fastflow's design employs a layered structure, with each layer building upon the previous one. The foundation layer includes thread-safe, wait-free Single Producer Single Consumer (SPSC) FIFO queues for synchronization in multithreading. The intermediate layer incorporates lock-free SPMC, MPSC, and MPMC queues, created using the SPSC queues from the foundational layer to facilitate communication between graph nodes generated by skeleton libraries. The top layer, visible to library users, offers various parallel skeletons (such as pipe, farm, and divide & conquer) represented as graph nodes where each node operates independently in its thread. These nodes utilize queues from the lower layer for communication, connecting different nodes in the graph.

For parallelism on clusters of multicore workstations, Fastflow's implementation involves two distinct approaches mentioned in online resources. Aldinucci et al. describe using ZeroMQ as the communication library to implement distributed parallelism [5]. ZeroMQ's asynchronous messaging model enables the creation of complex message-passing networks. Fastflow integrates customizable data serialization mechanisms for efficient data transfer across the network. Alternatively, the Fastflow project's GitHub page suggests employing MPI as the distribution library, although specific integration details are not provided. Cereal is utilized for data serialization. The choice between ZeroMQ and MPI for the distribution layer in cluster-level parallelism is not explicitly defined, indicating potential uncertainty regarding the standardized implementation approach in Fastflow. In the distributed version, each node can run its parallel pattern, leveraging shared-memory parallelism.

SkePU Framework

Originally established in 2010, the SkePU Framework [17, 20, 19] was developed as a skeleton programming framework with the aim of simplifying parallel programming on heterogeneous systems [18]. Utilizing C++11, SkePU offers a versatile interface for expressing parallel computations using high-level patterns. Key components of SkePU include a source-to-source precompiler and a runtime library, working in tandem to efficiently execute applications across various computational hardware. The framework supports a variety of skeletons tailored for common parallel patterns such as Map, MapPairs, MapOverlap, Reduce, Scan, MapReduce, MapPairsReduce, and Call [18].

SkePU effectively harnesses node-level parallelism through different backends optimized for specific hardware configurations. These include a sequential CPU implementation for baseline reference, OpenMP for multi-core CPUs, CUDA for NVIDIA GPUs, and OpenCL for GPUs and other accelerators. Additionally, a Hybrid backend is available for distributing workload across OpenMP, CUDA, and OpenCL backends, while the Cluster backend caters to distributed parallelism on large-scale clusters and supercomputers by leveraging the StarPU runtime system and MPI for coordination [18].

Muesli Library

The Muesli Library [9] originated from the University of Münster, Germany, as a skeleton library intended to streamline the development of applications for distributed memory systems. Built on C++ and making use of MPI for communication, Muesli offers two sets of skeletons. In terms of data parallelism, Muesli presents structures like DistributedArray, DistributedMatrix, DistributedSparseMatrix, alongside skeletons like fold, map, scan, and zip [9]. For task parallelism, it provides skeletons such as Pipeline, Farm, and DivideAndConquer, as well as atomic building blocks for constructing more customized parallel structures [9].

Muesli adopts a programming style that closely mirrors the traditional sequential development model. Node-level parallelism is achieved through distributed data structures partitioned across processes, enabling data parallel skeletons to operate on these partitions concurrently. Task parallelism, on the other hand, is accomplished by creating nested combinations of the provided task parallel skeletons [9]. The distributed execution capabilities of Muesli heavily rely on MPI (Message Passing Interface) for managing communication and synchronization across the distributed system.

3.5 Summary

This chapter provides an overview of parallel patterns, their history, and existing parallel pattern libraries. It discusses the structured parallelism approach using algorithmic skeletons or parallel patterns, which emerged in the late 1980s to simplify parallel programming. The advantages of using parallel patterns, such as providing high-level abstractions and ensuring correctness, are outlined, along with potential drawbacks like limited flexibility. The chapter traces the historical evolution of parallel pattern frameworks through pioneering, colonization, and integration phases. It then examines case studies of various parallel pattern libraries for C and C++, including eSkel, SKElib, Fastflow, SkePU, and Muesli, highlighting their key features, design principles, and implementation approaches for shared-memory and distributed parallelism.

Chapter 4

Design

This chapter goes over the key design decisions made to implement the farm and pipeline patterns at the node-level and the distributed-level.

4.0.1 Node-Level Parallelism

At the node level, two parallel patterns are available: the farm and the pipeline patterns. To achieve parallelism at this level, a shared memory programming model is used. Shared memory programming aims to parallelize computation by employing multiple threads or processes that share the same address space.

The design of the parallel pattern library at this level is inspired by Fastflow [6]. Both the farm and pipeline patterns can be seen as nodes of computation that receive a stream of tasks as input, process these tasks, and produce a stream of results. Within each of these patterns (farm and pipeline nodes) are nodes responsible for executing the actual work. A node can be seen as a unit of execution or a thread that assists in performing the required work for the pattern. Effective communication between these nodes is crucial for their independent operation, which calls for the use of a shared data structure establishing communication channels between nodes. This shared data structure is in the form of a thread-safe Blocking Queue. The specifics of how these patterns work and communicate with each other will become clearer as we explore the farm and pipeline patterns in detail.

Node

A `Node` introduces an abstraction layer to simplify the implementation of farm and pipeline patterns, which is located in *Node.hpp*. Each node has an input queue, an output queue, and a task function which can be user-defined or provided by the skeleton. Nodes can be started to create a thread that initiates the node loop, and then joined to wait for an End of Stream

(EOS) signal indicating the completion of tasks. The node loop processes tasks by fetching from the input queue, performing computations, and placing results in the output queue. Once an EOS is encountered, the node exits its work loop and its thread terminates.

```cpp
class DoubleNode : public Node<void*> {
public:
    void* run(void* input) override {
        int i = (int) input;
        return (void*)(input * 2);
    }
};
```

Listing 4.1 Creating a Node that doubles an integer input

Example code showing the creation of a node that doubles an integer input is provided in Listing 4.1. This demonstrates the process of creating a node with a specific computation function and data type. The `void*` represents the type of elements that will be contained in the input and output queues of the Node. Our library does not provide type checking so nodes can receive and emit tasks of any type or `void*`.

With these abstractions in place, implementing parallel patterns becomes more manageable by creating multiple nodes operating in parallel and ensuring proper communication through queues or channels.

Farm Pattern

The farm comprises a specified number of worker nodes responsible for parallel computation. Worker nodes, created by extending the node class and implementing the `run` method, execute the actual computation done by the farm. Optionally, the farm may include an Emitter node and a Collector node. The Emitter node preprocesses input tasks before allocation to workers, while the Collector node performs post-processing of the results emitted by the workers. Both Emitter and Collector nodes are created by extending the Node class and implementing the `run` function.

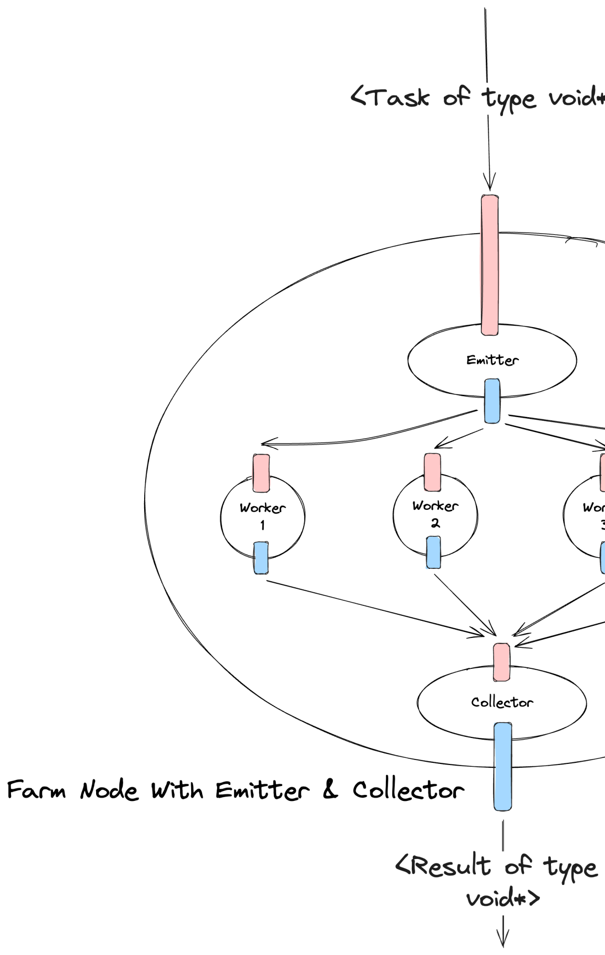


Fig. 4.1 FarmManager instance with an emitter and collector with 4 workers.

In Figure 4.1, we illustrate the logical arrangement of a farm consisting of an Emitter node, a Collector node, and four Worker nodes. The arrows demonstrate the flow of tasks and results within the farm, while the colored rectangles depict queues. Red represents input queues while blue is output queues. It is noteworthy that the input queue of the farm is also the input queue of the Emitter node, and similarly, the output queue of the Collector node and the farm node is shared. As the queues are thread safe this works well since having separate queues for these nodes would be quite wasteful of resource.

The code example accompanied with the output below shows how a simple farm can be created using the library. In this simple farm the factorial of a number is computed. We can notice how the work done by each part of the farm (Emitter, Collector, Worker) node is defined by creating nodes and providing it to the `FarmManger`.

```
// A simple node-level farm example where each worker computes a
    factorial of a number
#include <iostream>
```

```
#include "../../src/FarmManager.hpp"
#include "../../src/EOSValue.hpp"

using namespace std;

class FactorialEmitter : public Node<void*> {
public:
    FactorialEmitter(int num_tasks) {
        this->num_tasks = num_tasks;
    }

    void* run(void* _) override {
        if (curr == num_tasks) {
            return EOS;
        }
        std::cout << "Generated task " << curr << std::endl;
        curr++;
        return (void*) new int(curr);
    }

private:
    int num_tasks;
    int curr = 0;
};

class FactorialWorker : public Node<void*> {
public:
    void* run(void* task) override {
        int num = *((int*) task);
        long long result = 1;
        for (int i = 1; i <= num; i++) {
            result *= i;
        }
        return (void*) new long long(result);
    }
};

class FactorialCollector : public Node<void*> {
public:
    void* run(void* task) override {
        std::cout << "Factorial is " << *((long long*) task) <<
            std::endl;
        return task;
    }
```

```cpp
};

int main() {
    FarmManager<void*> *farm = new FarmManager<void*>();
    int num_tasks = 5;
    int num_workers = 5;

    FactorialEmitter *emitter = new FactorialEmitter(num_tasks);
    FactorialCollector *collector = new FactorialCollector();

    farm->add_emitter(emitter);
    farm->add_collector(collector);
    for (int i = 0; i < num_workers; i++) {
        FactorialWorker *worker = new FactorialWorker();
        farm->add_worker(worker);
    }

    farm->run_until_finish();
    delete farm;
    return 0;
}
```

Listing 4.2 A simple farm example in C++

```
rk76@bach:~/PPL/Node_Examples/Farm$ ../../build/farm_test
Generated task 0
Generated task 1
Generated task 2
Generated task 3
Generated task 4
Factorial is 1
Factorial is 2
Factorial is 6
Factorial is 120
Factorial is 24
```

One of the primary functions of the farm is to distribute tasks from the Emitter node's output queue to the Worker nodes' input queues. Currently, the farm employs a single load balancing strategy based on Round Robin to evenly allocate tasks among workers. While effective for tasks of equal size, more advanced load balancing strategies considering the workload of each worker could be developed in the future. The farm collects results from

Worker nodes and forwards them to the Collector node's input queues or the farm node's output queue if no Collector node is present. The following chapter will elaborate how End-of-Stream (EOS) symbols are managed.

Moreover, the farm supports nesting within other farms and pipelines at the node level, allowing for the creation of intricate parallel patterns. This nesting capability is enabled by the farm nodes (Emitter, Collector, and Worker nodes) extending the Node class.

Pipeline Pattern

In contrast to the farm pattern, the pipeline pattern involves parallel computation across several stages. Each stage receives input from the previous stage, imposing an ordered execution of computation. Similar to the farm pattern, each stage of the pipeline is created by instantiating a class extending the `Node` class and implementing the `run` method. Emitter and Collector nodes can serve as the first and last stages of the pipeline, respectively.

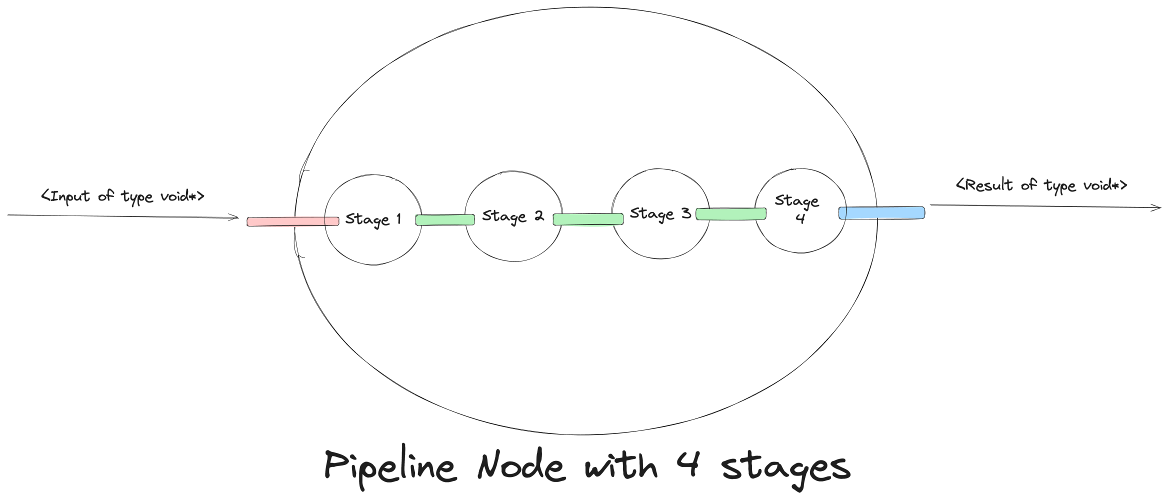


Fig. 4.2 PipelineManager Instance with four stages.

Figure 4.2 illustrates the logical layout of a pipeline node with four stages. Each stage is connected to the preceding stage through shared queues, facilitating data flow. Emitter and Collector stages share input and output queues with the pipeline node, avoiding use of any additional unnecessary queues.

The code example accompanied with the output below shows how a simple pipeline pattern can be created using the library. In this example the pipeline has two stages, a Generator and a Printer. The Generator emits tasks which are integer vectors, while the Print node prints the sum of every element in the array. We can notice how the work done by each stage of the pipeline is defined by creating nodes and providing it to the `PipelineManger`.

```cpp
// A simple node-level pipeline example with two stages
#include <iostream>
#include "../../src/PipelineManager.hpp"
#include <vector>
#include <numeric>

using namespace std;

class Generator : public Node<void *>
{
public:
    Generator(int num_tasks)
    {
        this->num_tasks = num_tasks;
    }

    void *run(void *) override
    {
        if (curr == num_tasks)
        {
            return EOS;
        }
        std::cout << "Generated task " << curr << std::endl;
        curr++;
        return (void *)new vector<int>(curr, 1);
    }

private:
    int curr = 0;
    int num_tasks;
};

class Print : public Node<void *>
{
public:
    void *run(void *task) override
    {
        vector<int> *v = (vector<int> *)task;
        std::cout << "Sum: " << std::accumulate(v->begin(), v->end
            (), 0) << std::endl;
        return nullptr;
    }
};

```

```cpp
int main()
{
    PipelineManager<void *> *pipeline = new PipelineManager<void
       *>();
    Generator *generator = new Generator(10);
    Print *print = new Print();

    pipeline->add_stage(generator);
    pipeline->add_stage(print);

    std::cout << "Number of stages: " << pipeline->
       get_num_pipeline_stages() << std::endl;

    pipeline->run_until_finish();
    return 0;
}
```

Listing 4.3 A simple pipeline example in C++

```
rk76@bach:~/PPL/Node_Examples/Farm$ ../../build/pipeline_test
Number of stages: 2
Generated task 0
Generated task 1
Generated task 2
Generated task 3
Generated task 4
Generated task 5
Generated task 6
Generated task 7
Generated task 8
Generated task 9
Sum: 1
Sum: 2
Sum: 3
Sum: 4
Sum: 5
Sum: 6
Sum: 7
Sum: 8
```

```
Sum:  9
Sum:  10
```

Unlike the farm, the pipeline pattern does not involve load balancing or result collection responsibilities, as all nodes are connected through shared queues. Handling of EOS symbols is explained in the next Chapter. Pipeline nodes can be nested within other farm worker nodes or pipelines using the `Node` abstraction like the farm.

4.0.2 Distributed-Level Parallelism

In the context of distributed computing, two parallel patterns are utilized: the farm pattern and the pipeline pattern. Parallelism at this level is achieved through the distributed memory programming model discussed in Chapter 3, which involves message passing between multiple processes to share data. Since these processes do not have access to a common shared memory, they rely on message passing for inter-process communication. To facilitate this communication, the library utilizes a Message Passing Interface (MPI) implementation, which provides processes with a user-friendly communication interface. The decision to use MPI was influenced by successful implementations in parallel pattern libraries like eSkel [12] and SKElib [16], saving significant development time compared to creating a communication library from scratch using lower-level networking protocols like Fastflow [6] or Java pattern libraries that utilize RMI like Muskel and Alt.

Farm Pattern

The structure of the farm pattern at the distributed level aligns with the node-level farm, with each node of the farm (Emitter, Collector, and Worker nodes) functioning as individual MPI processes. Coordination among these processes is handled by a Master process (with MPI rank 0), which monitors the Collector process for the receipt of an `EOS` symbol to determine when to end the MPI farm. It is important to note that both the Emitter and Collector nodes are essential components, and users must define both nodes for the MPI farm to operate correctly. Currently, the library does not support nesting of MPI farms within other MPI farms due to the complexities associated with MPI usage in such scenarios. However, nesting of node-level farms and pipelines within workers of an MPI farm or stages of an MPI pipeline is supported.

```
// Multi-level farm example for computing factorial of a number
// Uses two node-level farms as workers in an MPI farm

// mpiexec -np 5 ./cmake-build-debug/nested_farm_in_mpi_farm
```

```

#include <iostream>
#include "../../src/MPIFarmManager.hpp"
#include "../../src/FarmManager.hpp"
#include <algorithm>
#include <boost/mpi.hpp>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>

using namespace std;

class FactorialEmitter : public Node<string>
{
public:
    FactorialEmitter(int num_tasks)
    {
        this->num_tasks = num_tasks;
    }

    string run(string _) override
    {
        if (curr == num_tasks)
        {
            std::cout << "Generator Done" << std::endl;
            return EOS;
        }
        std::cout << "Generated task " << curr << std::endl;
        curr++;
        return std::to_string(curr);
    }

private:
    int num_tasks;
    int curr = 0;
};

class FactorialWorker : public Node<string>
{
public:
    string run(string task) override
    {
        int num = std::stoi(task);
        long long *result = new long long(1);
        for (int i = 1; i <= num; i++)
        {
            *result *= i;
        }
        return std::to_string(*result);
    }
};

class FactorialCollector : public Node<string>
{
public:
    FactorialCollector(int num_tasks)
    {
        this->num_tasks = num_tasks;
    }

    string run(string task) override
    {
        long long num = std::stoll(task);
        std::cout << "Factorial is " << num << std::endl;
        receive_count++;
        if (receive_count == num_tasks)
        {
            cout << "Collector Sending EOS" << endl;
            return EOS;
        }
        return string("CONTINUE");
    }

private:
    int receive_count = 0;
    int num_tasks;
};

int main()
{
    mpi::environment env;
    mpi::communicator world;

    MPIFarmManager *mpi_farm = new MPIFarmManager(&env, &world);
    int num_tasks = 5;
    int num_workers = 2;

    FactorialEmitter *emitter = new FactorialEmitter(num_tasks);
    FactorialCollector *collector = new FactorialCollector(
        num_tasks);
    mpi_farm->add_emitter(emitter);
    mpi_farm->add_collector(collector);

    // Node farm 1
    FarmManager<string> *node_farm = new FarmManager<string>();
    for (int i = 0; i < num_workers; i++)
    {
        FactorialWorker *worker = new FactorialWorker();
        node_farm->add_worker(worker);
    }
    mpi_farm->add_worker(node_farm);

    // Add node farm 2
    FarmManager<string> *node_farm_2 = new FarmManager<string>();
    for (int i = 0; i < num_workers; i++)
    {
        FactorialWorker *worker = new FactorialWorker();
        node_farm_2->add_worker(worker);
    }
    mpi_farm->add_worker(node_farm_2);

    mpi_farm->run_until_finish();
    return 0;
}
```

Listing 4.4 A MPI Farm where each distributed node is a node-level farm

```
rk76@pc7-003-1:.../Documents/PPL/MPI_Examples/Farm $ mpirun -np 5
../../build/nested_farm_in_mpi_farm
Generated task 0
Generated task 1
Generated task 2
Generated task 3
Generated task 4
Generator Done
Factorial is 6
Factorial is 24
Factorial is 2
Factorial is 1
Factorial is 120
Collector Sending EOS
```

The code example and the output above shows the creation of MPI farm with nested node-level farm as MPIFarm workers. It computes factorials of numbers just like the node-level farm example shown earlier but it does it using through nesting of node-level farms within a distributed-farm.

Pipeline Pattern

The design of the pipeline pattern follows a similar approach to that of the farm pattern. A Master MPI process (with rank 0) waits for an `EOS` symbol from the last stage of the pipeline to terminate the MPIPipeline. Each stage of the pipeline operates as a separate MPI process. The pipeline allows for various nesting configurations of node-level farms and pipelines within each stage, similar to the farm pattern. However, nesting of other MPI farms or pipelines within each stage of the pipeline is not supported, similar to the MPI farm.

```cpp
// A simple distributed pipeline example with two stages

#include <iostream>
#include "../../src/MPIPipelineManager.hpp"
#include <vector>
#include <numeric>
#include <sstream>
#include <boost/mpi.hpp>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>

using namespace std;
namespace mpi = boost::mpi;

class Generator : public Node<string>
{
public:
    Generator(int num_tasks)
    {
        this->num_tasks = num_tasks;
    }

    string run(string task) override
    {
        if (curr == num_tasks)
        {
            std::cout << "Generator Done" << std::endl;
            return EOS;
        }
        std::cout << "Generated task " << curr << std::endl;
        curr++;
        vector<int> v(curr, 1);
        std::stringstream ss;
        boost::archive::text_oarchive oa(ss);
        oa << v;
        return ss.str();
    }

private:
    int curr = 0;
    int num_tasks;
};

class Print : public Node<string>
{
public:
    string run(string task) override
    {
        vector<int> v;
        std::stringstream ss(task);
        boost::archive::text_iarchive ia(ss);
        ia >> v;
        std::cout << "Sum: " << std::accumulate(v.begin(), v.end()
            , 0) << std::endl;
        return string("");
    }
};

int main()
{
    mpi::environment env;
    mpi::communicator world;
    MPIPipelineManager *pipeline = new MPIPipelineManager(&env, &
        world);
    Generator *generator = new Generator(10);
    Print *print = new Print();

    pipeline->add_pipeline_node(generator);
    pipeline->add_pipeline_node(print);

    pipeline->run_until_finish();
    return 0;
}
```

Listing 4.5 A simple MPI Pipeline example with two distributed stages (no nesting)

```
rk76@pc7-003-1:.../Documents/PPL/MPI_Examples/Pipeline $ mpiexec -np 3
Generated task 0
Generated task 1
Generated task 2
Generated task 3
Generated task 4
Generated task 5
Sum: 1
Generated task 6
Generated task 7
Generated task 8
Sum: 2
Sum: 3
Sum: 4
Sum: 5
Sum: 6
Sum: 7
Sum: 8
Generated task 9
Generator Done
Sum: 9
Sum: 10
Master done
```

The code example and the output above shows the creation of MPI pipeline that has two stages and computes the sum of every element in a vector that is generated by the first stage. It is similar to the example shown earlier for the usage of node-level pipeline pattern but in this example the pipeline is distributed.

4.1 Summary

This section addresses the key design decisions made during the implementation of the farm and pipline patterns at both the node and distributed levels. The following chapter will examine the interesting implementation details within the library.

Chapter 5

Implementation

This section will discuss the implementation details of the library, including the rationale behind certain choices and their implications.

5.1 Node-Level Parallelism

The library utilizes shared memory programming, employing pthreads (POSIX threads). The choice of pthreads over C++11 threads was driven by two reasons: firstly, pthreads offer a lower-level interface to threads, allowing for greater control, which was deemed important; secondly, pthreads offer better support for UNIX-like systems compared to C++11 threads, based on several articles reviewed.

The implementations for the node-level Farm pattern can be found in `src/FarmManager.cpp`, and the Pipeline pattern implementation is located in `src/PipelineManager.cpp`.

Blocking Queue

A thread-safe queue serves as the communication channel between two nodes, which can be found in the *Queue.hpp* file. The queue supports operations such as non-blocking `push`, blocking `pop`, non-blocking `try_pop`, and a non-blocking `size` operation. It is designed to work with any data type and employs class templates. The terms "blocking" and "non-blocking" refer to whether the thread will wait for an operation to complete or not.

The queue implementation utilizes a non-thread-safe queue from the C++ standard library (`std::queue`). Synchronization is achieved through a `pthread_mutex_t` and a condition variable `pthread_cont_t`, ensuring thread safety and signaling to other threads when new elements are available in the queue.

Farm Pattern

An important aspect of the implementation is that the `FarmManager` class is defined as an extension of the `Node` class to support nesting. This allows the workers of the Farm to be instances of either a Farm or Pipeline pattern, enabling nesting of patterns. While currently only two levels of nesting have been tested, the library theoretically supports any level of nesting for parallel patterns at the node level. The use of inheritance to represent the nesting of parallel patterns facilitated this functionality.

Additionally, the `FarmManager` class utilizes a template variable `T` to represent the serialization data type used for communicating messages between different nodes. Depending on whether the farm is being used as a nested node in an `MPIFarmManager` or as a node-level construct, `T` can be either `void*` or `string`. The choice of `void*` for node-level farms allows elements of any type to be stored in the queue, simplifying initial development with a focus on adding type checking features later. The necessity for a `string` serialization type is explained in the distributed-level parallelism section. The decision to use C++ templates was driven by the convenience they offer in implementing a static type checking system.

Handling of the `EOS` symbols is another key design choice. Two threads are utilized for task distribution to worker nodes and result collection, respectively. Additional threads are needed for distributing and collecting results concurrently. When the distribution thread receives an `EOS`, it pushes the `EOS` to the input queue of every worker node, causing them to stop their work loop and push the `EOS` task to their output queue. The collector thread tracks the number of `EOS` tokens collected from the output queues and terminates after receiving one `EOS` token per worker, ensuring each worker has shut down. This clean implementation of handling `EOS` tokens was seen as the most efficient approach for getting the implementation working.

Pipeline Pattern

The pipeline pattern has the same defining design decisions as the farm pattern but is more straightforward. The `PipelineManager` class serves as the pipeline and is a specialization of the `Node` class, allowing easy nesting of the pipeline pattern within other node-level patterns, similar to the `FarmManager` class.

Once again, the template variable `T` is used to represent the serialization type of each stage and the pipeline as a whole. The serialization type refers to the data type in which messages are transferred between stages. As with the farm pattern, the serialization type is expected to be `void*` for node-level pipelines for consistency.

Handling `EOS` symbols in the pipeline pattern is simpler than in the farm pattern and does not require additional distribution or collection threads. When a stage in the pipeline receives an `EOS` symbol, it exits its work loop (`pipeline_stage_loop` in `Node.cpp`) and passes the `EOS` to the next stage. Once the final stage of the pipeline completes its task, the entire pipeline pattern terminates. This method was seen as the most straightforward way to manage `EOS` tasks in the pipeline pattern.

5.2 Distributed-Level Parallelism

Several important implementation details are considered when implementing parallel patterns at the distributed level. These include mechanisms for serializing data when transferring between distributed nodes, selecting an appropriate MPI implementation, and the right MPI communication primitives.

5.2.1 Boost: MPI and Serialization

In the design phase, the necessity for an MPI-based message passing library was highlighted to support distributed memory parallelism. Boost, a collection of open-source C++ libraries, was chosen to address this requirement. Boost.MPI, a library for message passing in high-performance parallel applications, serves as a wrapper around existing MPI implementations (such as OpenMPI, MPICH2, and Intel MPI) to provide a more C++ friendly interface. Boost.MPI offers extensive support for user-defined data types and C++ Standard Library types, making it a suitable choice for this project. This decision was influenced by the limited datatype support in MPI and the desire for a more modern C++ development style.

Additionally, Boost provides a serialization library which facilitates the serialization and deserialization of custom data types or C++ objects. This library played a crucial role in the project as it took care of the data marshalling and unmarshalling steps that happen before and after the data is transferred over the network between distributed nodes.

5.2.2 Communication

MPI offers various communication methods, each impacting the performance of distributed applications differently. The library development employs blocking point-to-point communication, the simplest communication technique offered by MPI. This approach utilizes `boost::world.send` and `boost::world.recv` methods, which internally use `MPI_Send` and `MPI_Recv`, respectively.

While it is easier to implement, blocking communication may not be the most efficient approach as it requires the sender process to wait for the receiver process to be ready. Non-blocking point-to-point communication using `boost::world.isend` and `boost::world.irecv` used along with message buffers could have been a performance-enhancing alternative. However, the decision to use blocking communication was made to simplify the implementation process and create a functional prototype of the library.

5.3 Summary

To summarise, this section has examined key implementation details of the library at both the node and distributed levels. The discussion has highlighted certain weaknesses in the chosen implementation strategies and justified the decision-making process. Consideration of other potential implementations has been acknowledged, emphasizing the reasoning behind the selected approaches and their implications.

Chapter 6

Evaluation

In this section, we will evaluate the performance of the parallel pattern library by testing its usage on a specific set of examples. We will first analyze the speedups achieved using parallel patterns at the node level, then we will explore the key objective of the project, which is measuring speedups achieved by nesting node level patterns within distributed level patterns.

6.1 Performance

Performance improvements using node-level patterns and distributed-level patterns will be analysed in this section. All experiments in this section take samples of 5 readings for the sequential version of code solution (that is single-threaded) and the parallel version of the code solution (that makes use of a pattern). For each sample the speedup is measured by dividing the time taken for the sequential implementation by the time taken for the parallel implementation. The plots and tables summarise the mean speedups achieved and the standard deviations by varying parameters like the number of workers.

6.1.1 Node Level

At the node level, we will evaluate the performance speedups achieved by the farm and pipeline patterns using specific problems that can be parallelized through these patterns. The farm pattern experiments at the node level are conducted on a 28-core machine equipped with an Intel Xeon CPU E5-2690 v4, featuring 2 NUMA nodes each with 14 cores in total, supporting 2 threads per core. The pipeline pattern experiments are run on a computer in the CS lab with an Intel(R) Core(TM) i5-8400 CPU, which has a single socket with 6 cores. Detailed CPU and memory specifications can be found in Appendix 1.

Farm

To evaluate the performance of the farm pattern, speedups achieved by the farm pattern in solving two well-known farm parallelisable problems is analysed.

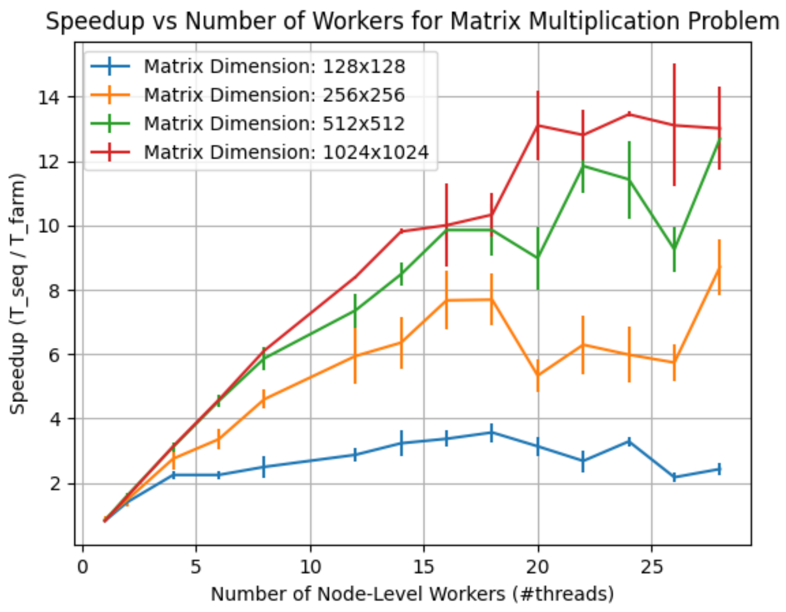


Fig. 6.1 Speedup achieved by using a node-level farm pattern for matrix multiplication.

Table 6.1 Speedup Data for Matrix Multiplication using a node-level farm

Matrix Dimension	Number of Workers	Mean Speedup	Std Dev Speedup
128x128	1	0.816	0.038
	2	1.406	0.149
	4	2.244	0.136
	6	2.244	0.132
	8	2.494	0.348
	12	2.870	0.197
	14	3.229	0.412
	16	3.368	0.248
	18	3.565	0.300
	20	3.134	0.301
	22	2.680	0.339

Table 6.1 – continued from previous page

Matrix Dimension	**Number of Workers**	**Mean Speedup**	**Std Dev Speedup**
	24	3.286	0.154
	26	2.172	0.136
	28	2.428	0.187
256x256	1	0.862	0.084
	2	1.485	0.214
	4	2.750	0.359
	6	3.354	0.316
	8	4.597	0.296
	12	5.939	0.867
	14	6.355	0.805
	16	7.671	0.914
	18	7.692	0.815
	20	5.333	0.519
	22	6.291	0.913
	24	5.982	0.872
	26	5.737	0.589
	28	8.709	0.859
512x512	1	0.829	0.035
	2	1.619	0.054
	4	3.119	0.152
	6	4.544	0.183
	8	5.868	0.359
	12	7.358	0.535
	14	8.480	0.363
	16	9.856	0.321
	18	9.854	0.815
	20	8.983	0.967
	22	11.847	0.824
	24	11.431	1.210
	26	9.253	0.702
	28	12.693	0.293
1024x1024	1	0.810	0.002
	2	1.587	0.012

Table 6.1 – continued from previous page

Matrix Dimension	Number of Workers	Mean Speedup	Std Dev Speedup
	4	3.134	0.011
	6	4.578	0.011
	8	6.110	0.035
	12	8.396	0.047
	14	9.809	0.086
	16	10.005	1.284
	18	10.334	0.684
	20	13.107	1.085
	22	12.811	0.802
	24	13.452	0.080
	26	13.110	1.909
	28	13.016	1.288

The first problem to solve using the farm pattern is matrix multiplication. The problem is set up such that two square matrices of the same dimensions, each containing a `double` element, are multiplied in parallel. The parallelization is achieved by dividing the first matrix into equal-sized partitions by rows, allowing the workers to compute the designated rows in the resulting matrix based on the assigned chunks of the first matrix. Since we are working in a shared memory setting, the pointers to the operand matrices and resulting matrix can be passed between the emitter, worker, and collector nodes. Figure 6.1 outlines the speedup achieved by using a node-level farm. It is observed that the speedup approached nearly 14 times as the number of farm workers increased to 28. Additionally, it is noticed that the speedups were significantly better for larger problem sizes. This outcome was expected since smaller problem sizes incur higher overhead in splitting tasks by the emitter node, collecting tasks by the collector node, and the work done by the distribution and collection threads, compared to sequentially solving the problem using fairly simple loops.

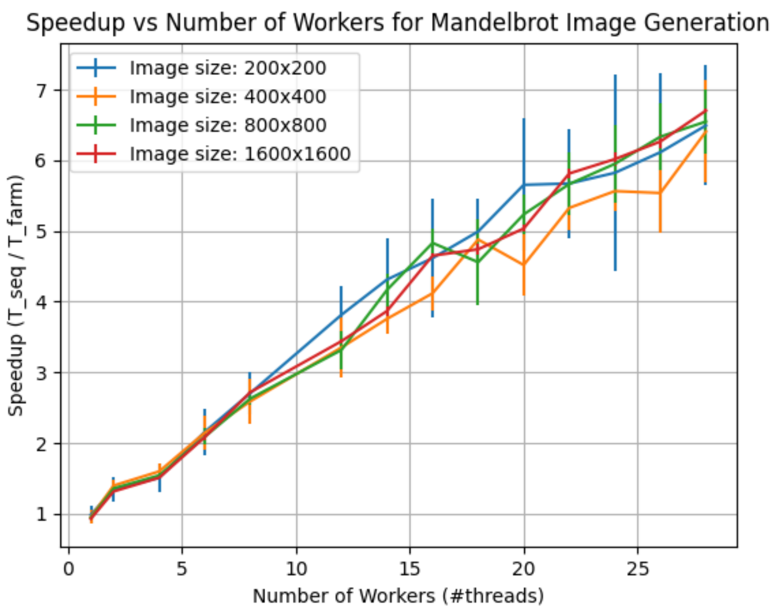


Fig. 6.2 Speedup achieved by using a node-level farm pattern for Mandelbrot Image generation.

Table 6.2 Speedup Data for Mandelbrot Image generation using a node-level farm

Matrix Dimension	**Number of Workers**	**Mean Speedup**	**Std Dev Speedup**
200x200	1	0.986	0.131
	2	1.346	0.178
	4	1.509	0.193
	6	2.158	0.324
	8	2.706	0.305
	12	3.814	0.409
	14	4.313	0.593
	16	4.616	0.833
	18	4.992	0.458
	20	5.653	0.934
	22	5.672	0.765
	24	5.823	1.387
	26	6.114	1.115
	28	6.493	0.848

Table 6.2 – continued from previous page

Matrix Dimension	Number of Workers	Mean Speedup	Std Dev Speedup
400x400	1	0.954	0.095
	2	1.398	0.086
	4	1.599	0.104
	6	2.146	0.236
	8	2.590	0.315
	12	3.353	0.428
	14	3.757	0.205
	16	4.119	0.235
	18	4.879	0.291
	20	4.518	0.436
	22	5.327	0.308
	24	5.564	0.281
	26	5.537	0.570
	28	6.406	0.722
800x800	1	0.944	0.015
	2	1.358	0.038
	4	1.542	0.027
	6	2.089	0.116
	8	2.630	0.096
	12	3.317	0.273
	14	4.166	0.227
	16	4.830	0.193
	18	4.559	0.604
	20	5.234	0.289
	22	5.664	0.448
	24	5.949	0.543
	26	6.333	0.466
	28	6.548	0.452
1600x1600	1	0.928	0.004
	2	1.315	0.016
	4	1.510	0.008
	6	2.084	0.048
	8	2.719	0.023

Table 6.2 – continued from previous page

Matrix Dimension	Number of Workers	Mean Speedup	Std Dev Speedup
	12	3.441	0.024
	14	3.868	0.071
	16	4.650	0.034
	18	4.742	0.075
	20	5.035	0.047
	22	5.813	0.076
	24	6.016	0.099
	26	6.265	0.094
	28	6.702	0.020

The second problem to parallelize is the computation of the Mandelbrot image. Farm parallelism is exploited by dividing the image into equal-sized chunks and allowing each worker to compute the designated chunk of the image. Figure 6.2 demonstrates the speedups achieved by the farm in comparison to the sequential solution of the Mandelbrot example, which runs on a single core. Speedups close to 7 times for most problem sizes was noted when workers went up to 28. The speedups for different problem sizes appeared quite similar, unlike the previous example, where larger problem sizes yielded better speedups with an increased number of workers. This could be because each task in this problem is a lot more computationally intensive compared to the task of multiplying two vectors together.

Pipeline

To illustrate the speedup achieved through the pipeline pattern at the node level, two examples are provided: a text processing pipeline and an image processing pipeline.

Table 6.3 Comparison of Sequential and Pipeline Processing Times (in seconds) for the Text Processing Pipeline

Iteration	Sequential Time	Pipeline Time	Speedup
1	6.1475	2.7536	2.2325
2	6.2386	2.6980	2.3123
3	6.1757	2.6859	2.2993
4	6.0648	2.6763	2.2661
5	6.2585	2.8852	2.1691
Average	**6.1770**	**2.7398**	**2.2559**
Std Dev	**0.0691**	**0.0775**	**0.0515**

The text processing pipeline consists of four stages. The first stage takes a folder path as input and reads every file in the folder, producing a vector of words. The second stage filters out stop words, the third stage counts the frequency of different words in each text file, and the final stage prints out the top three words in each file. The text data used in this experiment consists of 100 eBooks from Project Gutenberg in text format. Table 6.3 provides details on the speedups achieved by the pipeline pattern on a text processing example. A speedup of approximately 2.26 times is seen. The theoretical maximum speedup is around 4 since the pipeline has four stages, and the achieved speedup values are quite close to this, which is an encouraging result.

Table 6.4 Comparison of Sequential and Pipeline Processing Times (in seconds) for the Image Processing Pipeline

Iteration	Sequential (s)	Pipeline (s)	Speedup
1	3.1401	2.9898	1.0503
2	3.1646	2.9730	1.0645
3	3.1534	2.9974	1.0521
4	3.2119	2.9579	1.0859
5	3.1446	2.9814	1.0547
Average	**3.1629**	**2.9799**	**1.0615**
Std Dev	**0.0259**	**0.0137**	**0.0131**

The next problem uses the pipeline pattern to perform some image processing. Similar to the previous pipeline, this one also contains four stages: the first stage reads images from a given folder, the second stage converts the image to grayscale, the third stage applies a Gaussian blur (three passes), and the final stage writes the image to an output folder. Table 6.4 shows a speedup of only 1.06 times. This is because one stage in the pipeline is significantly more computationally expensive than the other stages. In this case, the Gaussian blur stage incurs the majority of the computation time. This example suggests that for optimal or near-optimal speedups to be achieved, each stage of the pipeline should consume approximately the same amount of computational resources. In other words, if one stage is much more computationally expensive compared to the other stages, it will block the progress of tasks in the subsequent stages, thereby reducing the impact of speedups achieved in previous stages of the pipeline (assuming speedups are achieved with the previous stages).

Table 6.5 Comparison of Sequential and Pipeline Processing Times (in seconds) for the Image Processing Pipeline with the Gaussian Blur stage being farmed with 6 workers

Iteration	**Sequential (s)**	**Pipeline (s)**	**Speedup**
1	3.1626	0.9069	3.4873
2	3.1319	0.9203	3.4030
3	3.1607	0.8868	3.5641
4	3.1424	1.7161	1.8311
5	3.0944	1.0872	2.8463
Average	**3.1384**	**1.1035**	**3.0264**
Std Dev	**0.0248**	**0.3146**	**0.6488**

To address the issue of the expensive Stage 3, a farm at that stage was introduced to distribute tasks among 6 workers. The choice of 6 workers was made to align with the lab machines, which have 6 cores, allowing each worker thread to be scheduled on a different core. By farming the third stage, significant performance improvements of around 3 times was observed, as seen in Table 6.5.

6.1.2 Distributed Level

Experiments at the distributed level are conducted primarily on a cluster of laboratory machines, where each lab machine is similar to the one used in the node-level pipeline experiments. The primary objective at this level is to assess performance enhancements by incorporating node-level patterns within a distributed-level pattern.

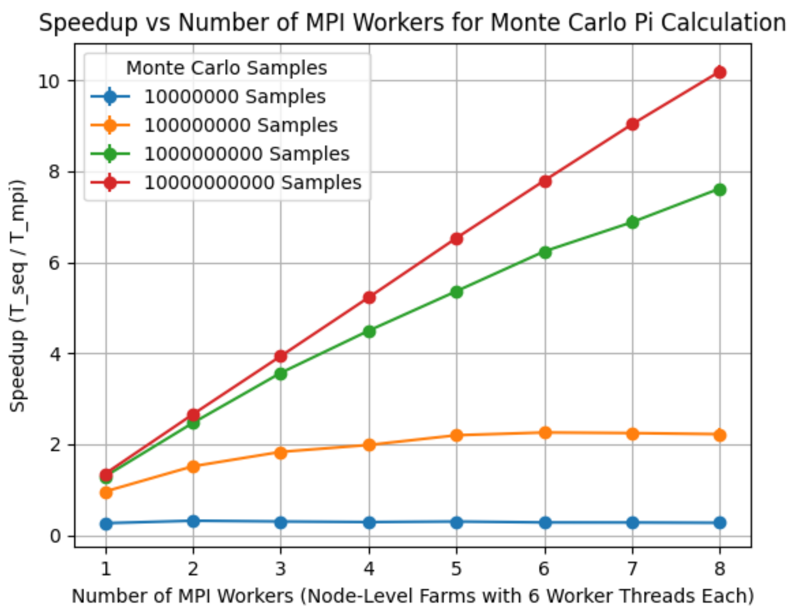


Fig. 6.3 Speedup achieved by nesting node-level farms within workers of an MPI farm for Monte Carlo PI estimation. Each MPI worker is mapped to a different lab machine with 6 cores each.

Table 6.6 Speedup Data for Monte Carlo Simulations using multi-level farm

Num Simulations	**Num Distributed Farm Nodes**	**Mean Speedup**	**Std Dev Speedup**
10,000,000	1	0.264	0.019
	2	0.317	0.013
	3	0.301	0.007
	4	0.289	0.008
	5	0.300	0.012
	6	0.280	0.009
	7	0.279	0.005
	8	0.273	0.009
100,000,000	1	0.958	0.039
	2	1.514	0.026
	3	1.831	0.066
	4	1.983	0.070
	5	2.197	0.114

Table 6.6 – continued from previous page

Num Simulations	Num Distributed Farm Nodes	Mean Speedup	Std Dev Speedup
	6	2.258	0.096
	7	2.245	0.088
	8	2.221	0.143
1,000,000,000	1	1.290	0.013
	2	2.467	0.030
	3	3.567	0.036
	4	4.494	0.041
	5	5.364	0.037
	6	6.240	0.068
	7	6.883	0.154
	8	7.624	0.063
10,000,000,000	1	1.346	0.006
	2	2.656	0.015
	3	3.936	0.013
	4	5.225	0.041
	5	6.533	0.038
	6	7.795	0.053
	7	9.036	0.039
	8	10.199	0.129

In order to assess the performance of the library, we utilize an embarrassingly parallel problem (problems that trivial to parallelise) of calculating the value of PI through Monte-Carlo simulations. The variable T_seq represents the time taken by a sequential version to compute PI using a specific number of Monte Carlo simulations with a single thread of computation. T_mpi indicates the time taken by the MPI farm, where workers are node-level farms running on different lab machines.

Figure 6.3 demonstrates the speedups achieved by adjusting the number of Monte Carlo simulations for estimating PI. It is evident that larger problem sizes yield improved speedups, similar to the behavior observed at the node level farm. Speeups close to 10 times is measured when using 8 distributed nodes that are farms with 6 worker threads each.

Notably, the overhead involved in distributing the data over the network for the MPI farm results in a more significant computational expense compared to the shared memory case where pointers to data items can be shared. This overhead is particularly noticeable when splitting the problem into parts and distributing them across different nodes, only making it more beneficial for larger problem sizes.

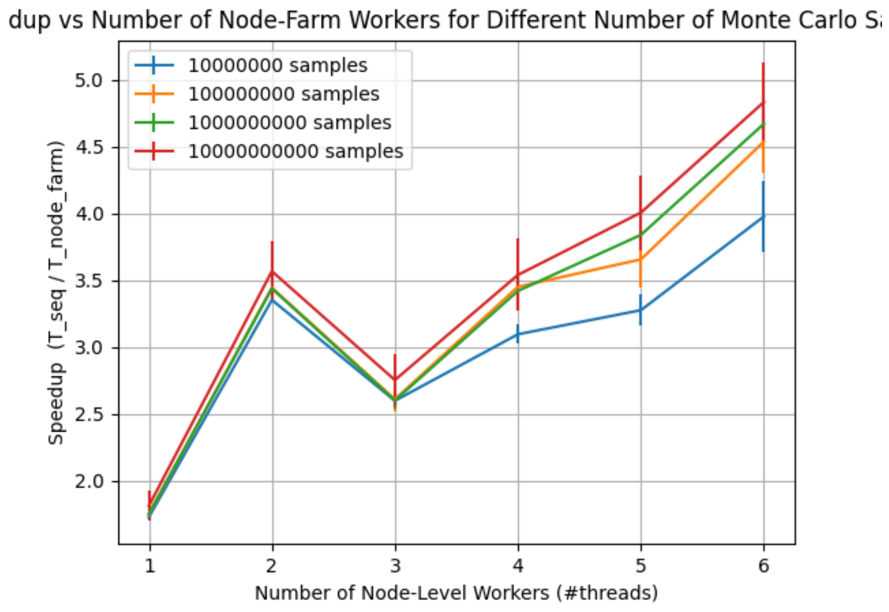


Fig. 6.4 Speedup achieved by using a single node-level farms pattern with 6 worker threads for Monte Carlo PI estimation.

Table 6.7 Speedup Data for Monte Carlo Simulations performed with a single node-level Farm with 6 worker threads

Num Simulations	**Num Workers**	**Mean Speedup**	**Std Dev Speedup**
10,000,000	1	1.730	0.012
	2	3.353	0.019
	3	2.600	0.053
	4	3.097	0.072
	5	3.277	0.119
	6	3.977	0.268
100,000,000	1	1.751	0.001
	2	3.444	0.004
	3	2.610	0.093
	4	3.452	0.055

Table 6.7 – continued from previous page

Num Simulations	Num Workers	Mean Speedup	Std Dev Speedup
	5	3.656	0.207
	6	4.537	0.231
1,000,000,000	1	1.754	0.002
	2	3.439	0.011
	3	2.602	0.061
	4	3.421	0.140
	5	3.840	0.045
	6	4.670	0.092
10,000,000,000	1	1.814	0.115
	2	3.568	0.226
	3	2.754	0.198
	4	3.540	0.270
	5	4.006	0.280
	6	4.833	0.292

Figure 6.4 shows a speedup close to five times is observed when a single node-level farm with 6 worker threads is used. Comparing this to the nested multi level farm 6.3 that yields a speedup of around to 10 times, which does not feel that significant because this case uses 48 cores in total spread across 8 lab machines which is 8 times the hardware resources. This suggests that for many problems, node-level patterns may be adequate, as there is substantial overhead in distributed-level parallelism that may not be justified by the amount resources required to setup distributed level parallelism.

6.2 Correctness

The library's correctness is currently validated through manual inspection of the outputs from running each example provided within the library, located in the folders `Node_Examples/` and `MPI_Examples`. There is currently no formal or automated testing in place to verify if the library is generating correct parallel code. Correct parallel code is defined as producing identical results to equivalent sequential programs. This area is a critical aspect of the library that requires attention.

6.3 Conclusion

The experiments have demonstrated that the parallel pattern library can significantly enhance performance in comparison to sequential implementations for embarrassingly parallel problems. Noticeable speed improvements have been observed both at the node-level and distributed-level. However, it is important to highlight several key considerations. Speed enhancements in scenarios utilizing the pipeline pattern are heavily dependent on the computational workload in each stage. If one stage necessitates a substantial amount of computation, the speed benefits achieved may diminish as it becomes a bottleneck impeding the progression of tasks to subsequent pipeline stages. When it comes to farm patterns, it is not advantageous to parallelize small problem sizes due to the increased overhead in data transfer between nodes. The multi-level farm pattern utilized in the Monte Carlo PI calculation example resulted in a speed increase twice as fast as the node-level farm pattern, despite employing eight times as many cores. This underscores the presence of considerable hidden overhead in network-based communications associated with distributed-level parallelism. It is important to note that these conclusions are based on a single instance of a multi-level farm pattern being utilized, and further research will be necessary to confirm these findings.

Chapter 7

Conclusions and Future Work

The project involved the development of a prototype parallel pattern library supporting parallel patterns at both the node-level (utilising shared memory parallelism) and at the distributed level (utilising distributed memory parallelism). The library supports farm and pipeline patterns at both levels, with nesting capabilities of node-level patterns within other node-level or distributed-level patterns.

A significant achievement of the project was the demonstrated performance enhancements achieved by nesting node-level patterns within distributed-level patterns, illustrating the potential of multi-level parallel patterns in addressing parallel computing challenges. The performance improvements seen in node-level patterns also confirm the effectiveness of shared memory parallelism. Some discussion considering the speedups to resource utilisation ratio were raised, questioning the need for multi-level patterns when node-level patterns offered very good performance improvements with a lot less resources.

Nevertheless, there are some drawbacks to be addressed. The limited number of examples showcasing the speedup achieved with multi-level patterns is insufficient to draw definitive conclusions on their utility, given their greater resource demands compared to node-level patterns. Additionally, the prototype library lacks support for certain features commonly found in modern parallel pattern libraries, such as static type checking, and the absence of automated testing poses a risk to the correctness of library results.

Looking ahead, if provided with additional time, efforts would be directed towards increasing the variety of examples to further validate the benefits of multi-level parallel patterns through speedup measurements. In the medium term, expanding support for parallel patterns like map and reduce would enable the library to address a wider range of parallelizable problems. In the long term, integrating support for exploiting multi-level parallel patterns on heterogeneous architecture machines such as GPUs or FPGAs could open up new opportunities for performance optimization.

This project has underscored the challenges of developing correct and efficient parallel code, reinforcing the importance of parallel pattern libraries. Debugging parallel code, particularly at the distributed level, remains a formidable task due to the lack of effective debugging tools, contrasting with the relatively simpler debugging process at the node-level using tools like **gdb**.

References

[1] Fastflow tutorial. Accessed on 25/01/2024. URL: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:tutorial.

[2] *pipe(2) — Linux manual page*, 2023.

[3] *pthreads(7) — Linux manual page*, 2023.

[4] $shm_overview(7)^{-}Linuxmanualpage$, 2023.

[5] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In Ioannis Caragiannis, Michael Alexander, Rosa Maria Badia, Mario Cannataro, Alexandru Costan, Marco Danelutto, Frédéric Desprez, Bettina Krammer, Julio Sahuquillo, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2012: Parallel Processing Workshops*, pages 47–56, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley Sons, Ltd, 2017. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13, doi:10.1002/9781119332015.ch13.

[7] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009. doi:10.1109/MSP.2009.934110.

[8] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[9] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The münster skeleton library muesli: A comprehensive overview. 01 2009.

[10] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. *Int. J. Supercomput. Appl.*, 8, 05 1996. doi:10.1007/978-3-0348-8534-8_21.

[11] Murray Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation*. 1988. URL: https://era.ed.ac.uk/handle/1842/11997.

[12] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004. URL: https://www.sciencedirect.com/science/article/pii/S0167819104000080, doi:10.1016/j.parco.2003.12.002.

[13] cplusplus.com. std::thread - c++ reference. http://www.cplusplus.com/reference/thread/thread/. Accessed: March 24, 2024.

[14] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998. doi:10.1109/99.660313.

[15] Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio Gonzalez-Velez, and Peter Kilpatrick. Algorithmic skeletons and parallel design patterns in mainstream parallel programming. *International Journal of Parallel Programming*, 49:1–22, 04 2021. doi:10.1007/s10766-020-00684-w.

[16] Marco Danelutto and Massimiliano Stigliani. Skelib : Parallel programming with skeletons in c. volume 1900, pages 1175–1184, 08 2000. doi:10.1007/3-540-44520-X_166.

[17] Johan Enmyren and Christoph W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, HLPP '10, page 5–14, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1863482.1863487.

[18] August Ernstsson. *Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems*, volume 1886. LinkÖping University Electronic Press, 2020.

[19] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. *International Journal of Parallel Programming*, 49:1–21, 12 2021. doi:10.1007/s10766-021-00704-3.

[20] August Ernstsson, Lu Li, and Christoph Kessler. Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *Int. J. Parallel Program.*, 46(1):62–80, feb 2018. doi:10.1007/s10766-017-0490-5.

[21] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, nov 2010.

[22] Sergei Gorlatch and Murray Cole. *Parallel Skeletons*, pages 1417–1422. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_24.

[23] John L. Gustafson. *Moore's Law*, pages 1177–1184. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_81.

[24] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Common Parallel Patterns*, pages 323–352. Apress, Berkeley, CA, 2021. doi:10.1007/978-1-4842-5574-2_14.

[25] Arch D. Robison. *Intel® Threading Building Blocks (TBB)*, pages 955–964. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_51.

[26] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[27] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330020404, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4330020404, doi:10.1002/cpe.4330020404.

[28] CORPORATE The MPI Forum. Mpi: a message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, page 878–883, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/169627.169855.

[29] D W Walker. Standards for message-passing in a distributed memory environment. 8 1992. URL: https://www.osti.gov/biblio/10170156.

Appendix A

User Manual

The section below provides information on how to run experiments to recreate some of the plots displayed in the evaluation chapter.

Building

- Build all targets by entering the following command:
  ```
  $ cmake -build <build-dir> -target all
  ```

Running experiments

- To run an experiment **cd** into the appropriate example folder under `Node_Examples/Farm` or `Node_Examples/Pipeline` or `MPI_Examples/Farm` or `MPI_Examples/Pipeline` depending on whether Node level or Distributed experiments are of interest

- Run the python script for the required experiment. This should produce two files. A JSON file with data relating to speedups for the experiment and a png file that should display the speedup curves.

Appendix B

Experiment Machine Specifications

B.1 Bach - 28 core shared memory Machine

```
rk76@bach:~/PPL$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         46 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  56
  On-line CPU(s) list:   0-55
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz
    CPU family:          6
    Model:               79
    Thread(s) per core:  2
    Core(s) per socket:  14
    Socket(s):           2
    Stepping:            1
    CPU max MHz:         3500.0000
    CPU min MHz:         1200.0000
    BogoMIPS:            5200.27
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   896 KiB (28 instances)
  L1i:                      896 KiB (28 instances)
  L2:                       7 MiB (28 instances)
  L3:                       70 MiB (2 instances)
NUMA:
  NUMA node(s):             2
  NUMA node0 CPU(s):        0-13,28-41
  NUMA node1 CPU(s):        14-27,42-55

# Memory Specs
rk76@bach:~/PPL$ free -h
               total        used        free      shared  buff/cache
available
Mem:           251Gi       8.2Gi       166Gi       0.0Ki
76Gi       241Gi
Swap:          8.0Gi        16Mi       8.0Gi
```

B.2 Lab machines - 6 core shared memory Machine

```
rk76@pc7-003-l:/cs/home/rk76/Documents/PPL $ lscpu
Architecture:               x86_64
  CPU op-mode(s):           32-bit, 64-bit
  Address sizes:            39 bits physical, 48 bits virtual
  Byte Order:               Little Endian
CPU(s):                     6
  On-line CPU(s) list:      0-5
Vendor ID:                  GenuineIntel
  Model name:               Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz
    CPU family:             6
    Model:                  158
    Thread(s) per core:     1
    Core(s) per socket:     6
    Socket(s):              1
    Stepping:               10
    CPU max MHz:            4000.0000
    CPU min MHz:            800.0000
    BogoMIPS:               5599.85
```

```
 Virtualization features:
   Virtualization:          VT-x
Caches (sum of all):
   L1d:                     192 KiB (6 instances)
   L1i:                     192 KiB (6 instances)
   L2:                      1.5 MiB (6 instances)
   L3:                      9 MiB (1 instance)
NUMA:
   NUMA node(s):            1
   NUMA node0 CPU(s):       0-5

# Memory
rk76@pc7-003-l:/cs/home/rk76/Documents/PPL $ free -h
                  total        used        free      shared  buff/cache
 available
Mem:               30Gi       4.7Gi        19Gi       185Mi
7.7Gi          26Gi
Swap:              15Gi       878Mi        14Gi
```